NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THE FORMAL SPECIFICATION OF A
VISUAL DISPLAY DEVICE:
DESIGN AND IMPLEMENTATION

by

James E. Hunter

June 1985

Thesis Advisor: Daniel L. Davis

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* The Formal Specification of a Visual Display Device: Design and Implementation		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis June 1985
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR*(s)* James E. Hunter		8. CONTRACT OR GRANT NUMBER*(s)*
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93943		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93943		12. REPORT DATE June 1985
		13. NUMBER OF PAGES 238
14. MONITORING AGENCY NAME & ADDRESS*(If different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution is unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Abstraction, specification, algebraic, interface, resource, image data type, point block transfer, bit-mapped display, portability

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

The visual display is usually treated as a separate I/O device. The interface to the programmer is at a low conceptual level and vaguely defined. Software that uses sophisticated displays are notoriously non-portable. In this study, we apply techniques using an axiom specification method to design, specify, and implement the resources of a bit-mapped color display device which is fully integrated with an abstract processor (Continued)

DD ₁ FORM JAN 73 1473   EDITION OF 1 NOV 65 IS OBSOLETE
S N 0102-LF-014-6601

ABSTRACT (Continued)

called AM. In conjunction, we provide a precise and high con-
ceptual interface to the resource to facilitate image programming.

S/N 0102- LF- 014- 6601

The Formal Specification of a Visual Display Device:
Design and Implementation

by

James E. Hunter
Lieutenant Commander, United States Navy
B.S., Pennsylvania State University, 1973

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1985

ABSTRACT

The visual display is usually treated as a separate I/O device. The interface to the programmer is at a low conceputal level and vaguely defined. Software that uses sophisticated displays are notoriously non-portable. In this study, we apply techniques using an axiom specification method to design, specify, and implement the resources of a bit-mapped color display device which is fully integrated with an abstract processor called AM. In conjunction, we provide a precise and high conceptual interface to the resource to facilitate image programming.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to thank the following people:

- Professor David L. Davis for his encouragement and guidance.
- My wife, Lydia, and my children, Bethany and Todd, for their great patience, understanding, and wonderful love.

I. INTRODUCTION

This thesis investigates the characteristics of the bit-mapped display device in order to define, specify, and implement a useful abstraction of this type of resource. A formal specification methodology is demonstrated that captures this abstraction in a precise yet readable manner. This research is an extension of the work begun by Yurchak (1984). That work dealt with the specification and implementation of an abstract machine we call AM.

AM (version 1.0) is the realization of an abstract processor together with its memory and primitive data types. AM (version 2.0) is derived from version 1.0 with the addition of the data types necessary to represent the abstraction of the bit-mapped display resource. It should be noted that the display resource is not viewed in this work as a *peripheral device* but is considered to be an integral part of the processor.

This research has two goals:

- Investigate and formulate an abstraction of the bit-mapped display resource with a useful set of abstract data types and operations.
- Demonstrate the feasibility of using the resource specification method to precisely describe a computing system by extending an existing processor's resource specification to include the display resource.

The following is a modification of the introduction presented by Yurchak (1984). The background and motivation for this research remains unchanged from that study.

A. THE PORTABILITY PROBLEM

It is well known that porting large programs from one machine to another is an expensive ordeal. It is also well known that once the software has been moved to the new machine, it is anybody's guess whether or not it will work as before[1]. Even if our program seems to work, we may find it consumes more resources than we expected. Indeed, this may be just as bad as if it did not work at all.

---

[1] We assume, probably unjustifiably, that it worked correctly before we tried to move it.

9

There are a number of reasons why the portability problem is getting worse, not better:

- Most architectures, even those which profess to be "language directed", reflect a bias toward making the machine look like what the programmer wants, or toward some engineering goal, such as maximizing the number of devices.
- Both languages and machines are related to the data they manipulate in an implementation dependent way.
- Language and hardware designers pursue their conflicting goals to the detriment of the poor compiler writer, who, with imprecise tools and methodologies is faced with the job of implementing ambiguous semantics on an informally designed resource.

Although these and other factors do adversely contribute to the imperfect task of moving software from one machine to another, they add their weight to other difficult issues in language design, computer architecture and software engineering. This study confines itself to treating the issues surrounding the interaction between the programmer's view of the world as a problem, and the architect's view of the world as a resource.

1. Abstraction

Abstraction describes the separation of the defining properties of an object from other, unnecessary details about it. A programmer is primarily concerned with solving a problem. Appropriately, the tools at his disposal, programming languages, development aids, the programming environment, form a *problem solving abstraction*. The hardware (and some of the software) on which this problem solving abstraction is implemented, however, is an abstraction of a different sort. Addresses, registers, ports, most of the operating system service routines, all provide more or less efficient ways to manipulate the physical resources of the machine -- they form a *physical resource abstraction*.

The fuzzy area between these two abstractions, sometimes simplistically perceived as the boundary between hardware and software, exposes a number of shortcomings in language design and computer architecture collectively termed the *semantic gap*.

Proper resource abstraction is essential for effective work in the portability problem area. In Davis (1985), Davis notes that implementors have

been reasonably successful building systems which promote portability when the resource in question has a clear model or abstraction. The file system is the prime example. Many operating systems (OS), such as UNIX or CP/M, provide a uniform, abstract, functional interface to the services of this resource.

Other areas are more difficult to abstract. Processors and visual displays are examples. The inability to establish a meaningful abstraction has impeded the formation of standard functional interfaces to these resources. OS's generally do not provide a functional interface to either the processor or the display[2]. Programs which access these resources directly, simply are not portable. High level languages (HLL) partially fill the gap left by OS's for the processor resource. Unfortunately, the interface level is high enough to force many applications to bypass the HLL for efficiency. Special graphics packages that extend the operating system provide similar services for the display resource. But despite these efforts, the problem is far from solved. The lack of a formal means to specify the interface that the OS's, HLL's and graphics packages attempt to provide is a serious shortcoming that impedes portability.

2. The Semantic Gap

The semantic gap manifests itself anywhere a problem solving abstraction touches a physical resource abstraction. A detailed description may be found in Myers (1982). He observes that the semantic gap contributes to the cost of software development, software unreliability, inefficiency, complexity, and the distortion of programming languages. Certainly no single development or methodology will eliminate this problem.

Narrowing the semantic gap requires significant changes in the fundamentals of computer architecture and language design. We chose to concentrate on three factors which significantly contribute to this problem:

- Informally described semantics.
- Representation dependent data types.
- Arbitrarily designed instruction set architectures.

---

[2]Except in the most rudimentary way. OS function calls to the display are usually limited to character and string output.

The implication, of course, is that through increased formalism, the introduction of representation independent data, and a more thoughtful treatment of the instruction set, the semantic gap can be narrowed. The balance of this thesis is devoted to describing a methodology for doing just that.

B. THREE WAYS TO NARROW THE SEMANTIC GAP

1. Formalism

The benefits of formalism in the design process have been amply revealed in countless articles treating this issue from the standpoint of software engineering. Our concern will be limited to formalism as it applies to the specification of an abstraction. Various specification methodologies exist, many of which have been used with more or less success in projects of practical significance. But we caution the reader that by "formal" we mean a mathematical rigor rooted in proven theory. The idea of formalism as often applied to software engineering will not do here. A *formal specification* is a complete description of the meaning of an object. It forms the basis for an abstraction and is ultimately a bridge over the semantic gap.

The benefits of formalism in which we are most interested are:

- It provides a firm basis for proving our assertions about a specification and its implementation.
- It encourages a discipline on the part of the designer to be rigorously precise.
- It compels us to find ways of describing things which are representation (implementation) independent.

2. Representation Independence

Conventional machines force us, as programmers, to develop our own abstractions of data. At a time when we are most concerned with developing clean algorithms the architecture obligates us to worry about status registers and word length. Certainly someone must ultimately deal with these physical properties of the hardware, but this should not fall as an *obligation* the upon programmer. The programmer should be free to ignore unnecessary detail.

Displays are equally difficult. Often the programmer is forced to deal with display data at a very low level. In order to create his display, it may be necessary for him to work at the level of poking bits out the processor port to the

terminal. By defining data types that include objects which represent concepts appropriate to visual display processing, the programmer will be freed to work at a higher conceptual level.

We will attempt to minimize the dependence of data upon its representation through the use of *abstract data types*. Our notion of data is very general. It ranges from integers, to image objects, and to program instructions. Data type representation will be hidden and abstract operations will be provided in the same way as with traditional abstract data types. If these data types can be kept representation independent, then portability is aided.

3. Intent Expressive Resource Abstraction

Conventional architectures do not permit us to unambiguously express our intent in a program. Artificial data types combined with typical resource models force ambiguity and the overloading of data structures. Stack frames are a good example of this. The semantics of the frame combine those of an array and those of a stack. Meanwhile, the whole thing is implemented in memory, with the data types overlaid on an array of fixed length cells.

We claim that applying methods similar to those used to describe abstract data types, we can describe an abstraction of the physical resources of a machine which benefits not only from the formalism used to specify it, but also permits the implementor to clearly interpret the intent of programs written for it, thus improving the efficiency of implementations.

C. METHODOLOGY

The goal of this research is to contribute something of practical significance to the study of software portability by treating an area which has been largely ignored -- the design of a formal abstraction for the computing machine itself. We have innumerable high level programming languages, programming environments, graphics languages, database machines, file systems, operating system command interpreters, a whole host of different abstractions tailored to the task of providing us with just enough information to do everything we need to do, and nothing more. So why, then, have we failed to develop abstractions for the hardware resources, upon which we are so dependent, which are more than just a collection of registers, opcodes and some arbitrary rules about how

13

they interact. A more difficult but certainly more important task than actually defining the abstraction is developing a methodology for producing other resource abstractions.

Our method has been to take a naive approach toward all areas of the design and implementation process not directly related to the specification itself. We do this for two reasons. First, we can take for granted the large body of research in programming languages and computer architecture -- we are designing neither a language nor a processor, even though *ad hoc* examples were required to complete the implementation. Second, the research is intended to benefit programmers. Since it is unreasonable to expect those who may use this method to understand the theory behind the specification, the key to understanding the reasons for our design decisions lies in the way we coded it. Thus, cleverness has been eschewed in favor of clarity.

Our task in this thesis, then, is to examine a wide range of issues which impinge on the process of designing and implementing the specification of a machine, and then to describe how we went about actually doing it.

II. THEORY

The formal specification method used to specify the AM machine is based on algebraic semantics. In Davis (1985), it is argued that resource abstraction is very similar to data type abstraction. In fact, to describe the computing resource, requires specification of both physical resources and data types. Guttag (1980) states that to describe an abstract data type precisely, the specification must embody the *semantics* in addition to the *syntax*.

.A formal specification must meet the following criteria if it is to be useful Guttag(1980):

- It must be restrictive enough to ensure that nothing unacceptable to the specifier will meet the requirements imposed by the specification.
- It must also be sufficiently general to ensure that few, if any, acceptable entries are precluded.
- It must be usable. The people that must deal with the specification must be able to understand it.

The latter point has precluded previous methods that would be otherwise acceptable.

From Yurchak (1984), we note that to achieve true portability, we must be able to demonstrate the following properties in our implementation:

- The specified semantics *actually implemented* on the source machine are completely unambiguous.
- The implementation on the source machine is "correct".

Thus, our method of specification must be formal enough to permit proofs of correctness. Although the knowledgeable reader will know that the provability of usefully complex specifications has so far been unrealized, research conducted in parallel with this study (Griffin 1984) has given us reason to be optimistic.

Algebraic specifications meet the above criteria. Here we find a significant body of research already in place in the area of abstract data type specification. Goguen (1978) and Guttag (1978) treat this topic in great detail. We will not do

15

so here. Instead we give an overview of the important concepts of abstract data types, and direct the reader to the original works for a more in depth study of the underlying theory. Davis (1984) provides the theoretical basis for the resource specification method. Davis (1985) provides additional background but with an emphasis on practical issues.

A. ALGEBRAIC SPECIFICATION

An algebraic specification of an abstract data type consists of two parts: a syntactic specification and a semantic specification. The syntactic specification provides syntactic and type checking information. The semantic specification is a set of axioms that define the meaning of the operations by stating their relationships to each other.

1. Syntactic Specifications

The syntactic specification identifies the list of operations that are provided by an abstract data type. The number and types of the operands used by an operator is specified as well as the return type. The operators and operand types are usually given, what is hoped to be, meaningful names. Unfortunately, when a reader encounters a name he recognizes, he tends to assume that the object has the properties he associates with that name. At first glance, this type of specification seems to be adequate.

To illustrate this fallacy, consider the specification in Figure 2.1 of the type queue (of integers) from Guttag (1980). Note that this specification can be converted to represent type stack by replacing queue with stack, add with push, front with top, and remove with pop. It is now apparent that the perceived semantics of this example is conveyed solely by our personal understanding of words such as add, front, queue, etc. The importance of the semantic specification is now evident.

2. Semantic Specification

The semantics of the data type are precisely specified by a set of axioms. The specification language itself consists of five primitives: the functional composition, the equality relation (=), two distinct constants (true and false), and an unbounded supply of free variables. From these, other constructs such as the if-then-else can be created. Figure 2.2 illustrates how the axioms are used to

```
spec queuetype
is
        sort
                queue;
        primitive
        op
                new: → queue;
                add: queue,integer →queue;
                front: queue → queue;
                remove: queue → integer;
                isempty: queue → boolean;
end queuetype;
```

Figure 2.1: Example of an Incomplete Spec

```
spec bagtype
is
        sort
                bag;
        primitive op
                empty_bag: → bag;
                insert: bag,Int → bag;
                delete: bag,Int → bag;
                ismember: bag,Int → bool;
        axiom
        1)      ismember(empty_bag(),x) = false;
        2)      if x = y
                        then ismember(insert(s,x),y) = true;
                        else ismember(insert(s,x),y) = ismember(s,y);
        3)      delete(empty_bag(),x) = empty_bag();
        4)      if x = y
                        then delete(insert(b,x),y) = b;
                        else delete(insert(b,x),y) = insert(delete(b,y),x);
end bagtype;

spec settype
is
        sort
                set;
        primitive op
                empty_set: → set;
                insert: set,Int → set;
                delete: set,Int → set;
                ismember: set,Int → bool;
        axiom
        1)      ismember(empty_set(),x) = false;
        2)      if x = y
                then ismember(insert(s,x),y) = true;
                        else ismember(insert(s.x),y) = ismember(s,y);
        3)      delete(empty_set(),x) = empty_set();
        4)      if x = y
                        then delete(insert(s.x).y) = delete(s,y);
                        else delete(insert(s,x),y) = insert(delete(s,y),x);
end settype;
```

Figure 2.2: Specs for the Abstract Types **Bag** and **Set**

precisely describe the semantics. Notice that the *syntactic* specifications are identical. The *semantic* specifications are also identical except for the *then* clause of axiom 4. These specifications point out the similarities and isolate the one crucial difference between the two types. While some of these axioms may require some thought to realize what idea they are conveying, they are reasonably understandable to an experienced programmer.

One must exercise care to avoid interjecting inconsistencies into the axioms that would permit the axioms to be combined in ways that would produce results such as true() = false() which, of course, is nonsense. Additionally, the set of axioms must be complete enough to exclude any unacceptable implementations of the data type. The completeness problem is difficult one. It is beyond the scope of this discussion and the reader is referred to Guttag (1980).

3.   Error Handling

In any specification of interesting complexity, the user has the potential to combine operations in a sequence that causes an error. How the errors are handled is an important aspect of the specification. Two implementations of the

---

```
spec natural
is
        sort
                nat;
        primitive
        op
                zeronat: → nat;
                prednat: nat → nat;
                succnat: nat → nat;
        axiom
                prednat(succnat(n)) = n;
                succnat(prednat(n)) = n;
end natural;
```

Figure 2.3: A Simple Spec for **Natural**

19

specification can not be equivalent if they respond differently to errors. Consider the simple specification for *natural* in Figure 2.3 which is limited to only counting operations. The natural number "0" is represented by the constant[3] zeronat(). Since the natural number zero has no predecessor,

$$prednat(zeronat())$$

is obviously an error.

The following points are applicable to the error handling issue:

- Any operation which encounters an error is computationally meaningless.
- If an operation encounters an error, then any subsequent operation which utilizes the errant result must also return an error.
- Errors must not be hidden; they must be made known to the user.

One approach that was used in Yurchak (1984) and Goguen (1977), is to use a class of operators called error operators. With this method, the operator class:

error
op
naterror: → nat;

is added to the spec natural in Figure 2.3. The effect is to create a constant that represents each specific error. This constant is the same type that the operation normally returns and becomes the return value for the illegal operation. One advantage, is that the occurence of an error is not only recognized but the specific type is known.

On a practical level, this approach causes the number of axioms to explode for even moderately complex specifications involving error operators. Since naterror() is a legal nat type, it can be substituted into any free variable of type nat. The specification writer must now specify the meaning of using naterror() in each operation which uses nat as an operand. For our **spec natural** the axioms

---

[3]Really a O-ary operator.

$$\text{prednat(naterror())} = \text{naterror();}$$
$$\text{succnat(naterror())} = \text{naterror();}$$

must be added. Similar axioms are required for any other operations involving **nat**. The explosion really occurs when **spec natural** is extended by other specifications that introduce new sorts that also use **nat** operands.

For this reason, we have abandoned this approach and introduce the concept of *undefined*. *Undefined* has the following semantics:

- Any illegal operation is **undef**.
- If t = **undef** then $A(x_1, x_2, ..., t, ..., x_n) = undef$;

. where **A** is any operator in the specification, and x_i is an expression.

- Any equation involving **undef** is meaningless.
- Whenever **undef** is encountered, the processing halts immediately and an appropriate error message is issued.

This method keeps the number of additional axioms needed for errors manageable. Consider again our spec natural in Figure 2.3. Adding the axiom:

$$\text{prednat(zeronat())} = \text{undef;}$$

solves the predecessor of zero problem without introducing an error operator or constant.

The effect is to restrict to range of free variables that apply to an axiom. For instance, if **zeronat()** is substituted for the free variable **n** , then the axiom

$$\text{succnat(prednat(n))} = \text{n;}$$

evaluates to

$$\text{succnat(prednat(zeronat()))} = \text{zeronat();}$$
$$\text{succnat(undef)} = \text{zeronat();}$$
$$\text{undef} = \text{zeronat();}$$

This implies that the above axiom applies to all free variables except n = zeronat().

One short coming of this method is that different error handling philosophies can not be easily expressed. Goguen (1978) proposes several ways for handling this problem, but without much success for the practical application. Continued research in this area is needed.

It is important to understand the difference between *undefined* and *not implemented*. Obviously, any actual machine will be finite despite the fact that our specification includes several infinite sets of objects, **spec integer** for example. In fact, the processor resource is, itself, described as infinite. **Spec memaddress** contains a starting memory address for each memory segment but has no bound on the size of each segment nor even the number of segments! One might expect an axiom to describe that referencing a memory address beyond the last available address is undefined.

The memory size in the specification is not bounded for the same reasons that integer is unbounded. Conceptually, there is no natural limitation on the size of memory. Any implementation, of course, will be limited in memory and in the range of integers it can process. Any instruction which references a non-existent memory location will generate an appropriate error. The distinction is that such an instruction is *not implemented* rather than *undefined*. On a larger machine, the same instruction may be permitted. Note the difference from a truly undefined operation such as *pop(empty stack)* which is conceptually meaningless and an error in all implementations.

III. <u>DESIGN ISSUES</u>

Before the display resource can be specified, a useful abstraction of this resource must be formed. In order to formulate this abstraction, an understanding of the basic principles and environment under which the programmer operates is required. This abstraction should support the programmer and encourage him to work at a high conceptual level. He should think about and manipulate "image objects" rather than just dots and lines. A poor abstraction of the resource will doom the design from the outset. We now take a look at how current graphics systems approach the programming task.

A. BASIC GRAPHICS PRINCIPLES

One of the first problems we encounter in display programming on many systems, is the difference between a graphics and a character display. In many implementations, they are mutually exclusive operations. Conceptually, it depends on your point of view as to whether they are really the same or different. In the broadest sense, character display is just one instance to the more general graphics problem of creating images on a screen. The differences have arisen in an effort by implementations to speed up the very common task of character display. Graphics is inherently a computationally bound task when one considers the number of points on the screen which must be manipulated. Hardware is often provided to capitalize on the specialized characteristics of the character display. This enables text to be rapidly presented, moved and scrolled. But the general graphics problem is sometimes complicated by the non-regularity of this approach. Further, the existence of the character display hardware may actually hinder general graphics display. Our approach is to consider character display to be a subset of the more general problem of graphics display.

1. <u>A Simple Graphics Package</u>

To understand where and how the AM display resource fits into the overall scheme, we will describe a traditional graphics system. The system we will consider is taken from Foley (1984) and is called the *Simple Graphics Package*

(SGP). This system is representative of many existing packages. SGP is an interactive system which handles both input and output. We intend to simplify the system and discuss only the functional aspects of the output that are pertinent to this research. Figure 3.1 depicts our view of the conceptual block diagram for the SGP system.

The SGP model breaks software into three components, the application data structure, the application program, and the graphics system. The application data structure is used to model display objects. It holds the description that is needed to describe an object visually. The form of this information is usually geometric coordinate data, object attributes such as surface color, and connectivity relationships. The coordinate data of the model is usually expressed in *world coordinates* (WC) using whatever units are natural to the application. World coordinates are essentially unbounded.

The application program builds and maintains the object data structure. To view the object, the program interprets the object data structure and issues appropriate calls to the graphics system interface. These calls are graphic output
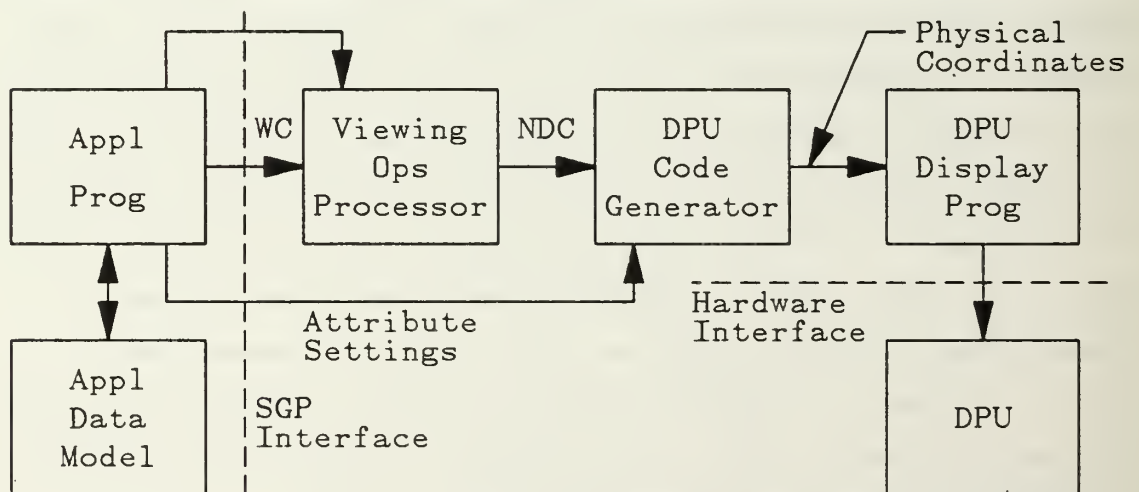
Figure 3.1: SGP Block Diagram

primitives. Graphics output primitives are essentially a high level language that is machine independent. Typical primitives are points, lines, and polygons expressed in 2D or 3D world coordinates.

Conceptually, SGP is divided into two major modules. The first is the *viewing operations processor*. This module transforms the object's visual image in a number of ways. It provides translation, scaling, rotation, and perspective operations. The application program provides this module with a specification that describes the region of interest and the desired perspective expressed in WC. Then, as the program submits its description of the object in the form of output primitives, the viewing device processor transforms the input to create the desired perspective.

The total transformation is a series of several smaller steps. For 2D graphics, the program specifies a *window* and *viewport*. As the SGP receives primitives from the program, they are checked against this window definition. Any portion of the primitive that falls outside the *window* is clipped. The image of the *window* is then mapped onto the *viewport*. The *viewport* is described in terms of the display screen. It may cover the entire screen or any rectangular portion. Each axis of the *window* is scaled to fully occupy the corresponding axis of the *viewport,* thus, the aspect ratio is not necessarily preserved during this mapping. Use of the translation, scaling and rotation operations, frees the application program from recomputing the object model in a new position. The desired operation is passed to the *viewing operations processor* and the original object description is submitted. The *viewing operations processor* makes the necessary adjustments. Thus, a series of transformations can be accomplished without changing the data structure representation of the model. Only the transformation specification must be modified.

Several additional operations are needed for 3D graphics. A *view volume* is specified in addition to the *window* and *viewport*. Clipping is performed against this volume. The mapping from 3D to 2D requires a projection operation. SGP provides several types of projections which fall into two basic classes: *perspective* and *parallel*. After the clipped volume is projected onto the *window,* the *window* is mapped onto the *viewport* as before.

With the exception of specifying the location and size of the *viewport* on the screen, all application program work is done in WC with no regard to the physical device which ultimately responds to the primitives. During the transformations, the *viewing operations processor* converts the WC to *normalized device corrdinates* (NDC) based on the viewport specification. NDC ranges from 0 to 1 relative to the physical device's screen boundaries. The transformed primitives which are now expressed in NDC are then passed to the second major module of SGP, the *display processor unit* (DPU) *code generator.*

The DPU *code generator* accepts NDC primitives from the *viewing operations processor,* which are still device independent, and converts them into a equivalent series of device-dependent hardware instructions expressed in the physical device coordinates for the the target DPU. *Attribute* primitives which do not involve coordinates are received by the *code generator* directly from the application program. Foley (1984) uses a paradigm to describe the DPU *code generator* as a "display program compiler" which generates code from a machine independent high level language into a low level code for a specific DPU. Note that the compilation of DPU code occurs during runtime as the application program issues output primitives.

   2.   The AM Display Resource

The AM display resource replaces three of the items in Figure 3.1, the DPU code generator, DPU display program, and the DPU. Figure 3.2 shows the block diagram for a graphics system using AM. AM's drawing primitives are slightly more low level than SGP and are limited to point, line, and fill rectangle. On the other hand, AM has several data structures which permit whole images to be manipulated, copied and moved. They are designed to encourage the programmer to think and work at a high conceptual level. The use of these data structures with their defined operations, give the programmer a very powerful instructions set.

Like the DPU code generator in SGP, the AM interface accepts primitives in NDC and is machine independent. Unlike SGP, the NDC range is from 0 to 9999 rather than 0 to 1. The AM NDC system is equivalent to SGP's

```
┌──────────┐        ┌──────────┐  WC ┌──────────┐ NDC ┌──────────┐
│   Appl   │        │          │     │ Viewing  │     │          │
│   Data   │◄──────►│   Appl   │────►│   Ops    │────►│    AM    │
│  Model   │        │   Prog   │     │Processor │     │          │
└──────────┘        └──────────┘     └──────────┘     └──────────┘
                         │                                   ▲
                         └───────────────────────────────────┘
                                   Attribute
                                   Settings
                            SGP              AM
                            Interface        Interface
```
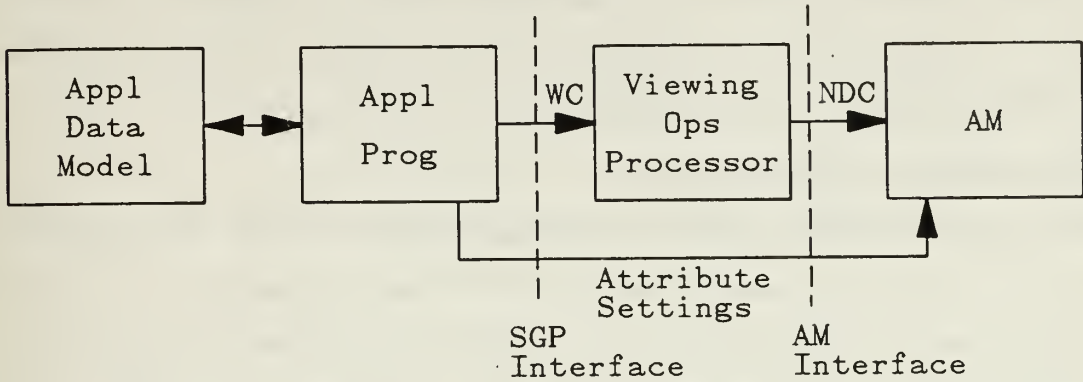
Figure 3.2: SGP Block Diagram on AM Resource

but is more convenient to capture the programmer's intent, a concept that will
be amplified later.

AM does have window and clipping operations. However, these
operations are conceptually quite different from the SGP primitives.
Conceptually, *images* may be larger that the display screen. AM's *window*
represents the fixed size of the display screen and can be thought of as a *framing
mat* that can be positioned over a picture that may or may not be larger than its
opening. With this idea and recalling that AM's input primitives are expressed
in NDC rather than WC, it should be apparent that AM is not intended nor
capable of transforming portions of a model expressed in WC. AM does not have
a projection capability nor translation, scaling, or rotation transformations.
These operations are not considered appropriate for this abstraction level. The
idea is to model the display resource, which is a flat screen that projects an
image. The construction of an image from the model of a 2D or 3D object is best
left to an application program running on SGP which in turn runs on AM. In
this scenario, the SGP would be used to create the machine independent image
which any AM hosted machine could faithfully display.

In general, most applications have no need to create images from a data structure representing an object. Applications of this type directly construct a screen image without the services of a graphics package. AM is very well suited for this type of application as well. In fact, the image data type permits a programmer to built complex images with minimal effort.

B. THE PHYSICAL RESOURCE

When one is asked to point to the visual display resource of a computing system, he is likely to indicate that the "TV" like component with the CRT screen is the central part of the resource. But from the programmers perspective, the bit-map which stores the image information is the heart. The CRT with its display screen is merely a *magnifying glass* that is passed over the memory containing the bit-map. The CRT allows us to *see* that portion of the memory contents under the *magnifying glass*. The programmer doesn't program the CRT. The image projected onto the screen is a side effect of memory manipulation. In summary, the display resource for our purposes, consists of the display memory plus any programmable memory cell or register, which controls attributes that affect the projected screen image.

In AM, *images* are considered abstract data types. Conceptually, image display is accomplished in the following way. The *image* object is initially stored in memory. To alter it, it is first fetched from its memory location. This *image* is then used as an operand in some *image operation*. The resultant *image* object is then stored back into memory. At any instance, the memory may contain several *image* objects. The terminal is directed to *view* a selected *image* in the context of the current terminal *attribute set*. This *attribute set* includes items such as *background color* and other items needed by the programmer. The *attribute set* also includes informational items concerning terminal characteristics that the programmer may need. such as the screen dimensions and the physical number of pixels. Applications would not normally alter terminal characteristics, but a configure type utility for an operating system would need to update these when hardware changes are made.

C. INTERFACES

The purpose of abstracting and specifying the computing resource is to permit construction of equivalent instantiations of the specification on different hardware resources. Aside from the limitations on the practicality of proofs of correctness, each implementation can be checked to be a *correct* instantiation of the specification. Such implementations are provably equivalent, Davis (1985). It is startling to realize that each of these equivalent implementations may have

---

```
        spec boolean
is
        sort
                bool;
        primitive
        op
                true: → bool;
                false: → bool;
                not: bool → bool;
                and: bool,bool → bool;
        derived
        op
                or: bool,bool → bool;
                implies: bool,bool → bool;
        derived
        def
                or(b1,b2) = not(and(not(b1),not(b2)) );
                implies(b1,b2) = not(and(b1,not(b2)));
        axiom
                false() = not(true());
                not(not(b)) = b;
                and(true(),b) = b;
                and(false(),b) = false();
                commutative(and,bool);
        end boolean;
```

Figure 3.3: A Spec for Abstract Type **Boolean**

---

different user interfaces! It turns out that these different interfaces are all equivalent and portability is not in jeopardy.

Consider the following to demonstrate this principle. Figure 3.3 shows the specification for boolean. The specification is defined in terms of *true, false, not* and *and*. The operators *or* and *implies* are derived from *not* and *and*. Assume instantiation **A** implements all the operators and instantiation **B** leaves out the derived operations. The student of logic knows that any boolean expression using the *or* or *implies* operators can be rewritten using only the *not* and *and*. A translator can be created which will take programs written for instantiation **A** and mechanically convert any references to *or/implies* into their *not/and* equivalent which is provably equivalent. If both the input programs and both the implementations are equivalent, then the outputs will also be equivalent. Even in the pathological case where **A** implements only *not/and* and **B** implements only *or/implies*, the interfaces remain equivalent.

In summary, the resource specification provides the basis on which implementations are judged to be equivalent. The implementor implements the specification which can (in theory) be tested for compliance. In the process of implementing the specification, the implementor has a degree of latitude in the choice of which operators from the specification are made available to the user. If he chooses too few or leaves out the wrong ones, his instantiation as well as the interface, will be provably incorrect. On the other hand, if more operators are put in the interface than the minimal set, the interface only becomes functionally redundant. His goal is to select a set which is functionally complete but not overly redundant.

IV. THE SPECIFICATION LANGUAGE

The specification language used to describe AM (version 2.0) is essentially the same as AM (version 1.0) by Yurchak (1984). However, our view of the best form for the language has continued to evolve with the passage of time and continued research. Chapter III of Yurchak(1984) discusses many pertinent issues relating to this subject. The reader is encouraged to review this material. Attention is directed to the sections on parameterized specifications and extensions.

A. THE GRAMMAR

Appendix A contains the grammar for the specification language. It is similar to examples found in literature but with changes that give it the flavor of a programming language. A practical specification language must support automated parsing. In Lilly (1984), this language was used as the demonstration language for a syntax directed editor. This editor accepts the grammar of a specification language and guarantees that any specification built will be syntactically correct.

Using this language, a specification is constructed with modules called *spec*. The *extension* operator is used to link the *specs* into a hierarchy. Each spec may introduce zero or more new *sorts, operators* and/or *axioms*. *Sorts* are similar to data types. They form object sets from which the operands are selected for the operators. The elements in a *sort* are created from *operations*. Usually, one or more constants are declared to provide the basis from which other elements are created. In **spec natural**, for example, **zeronat()** is the basis element. Other elements are generated from the application of the operators. Succnat(zeronat()) yields "1", succnat(succnat(zeronat())) yields "2", etc. Sorts introduced in a spec may be added to an existing spec through extension or may form the primitives for a new *branch* of the hierarchy. *Extension* provides the only means of relating the *sorts* and *operators* from different specs.[4]

---

[4]There are several other operations by which two specifications may be related. They are

For example, in the AM specification, **spec boolean** forms the root of the hierarchy. It introduces **sort bool** which represents the boolean abstract data type. The operations are declared in terms of sort arguments and return sorts. The elements of this sort are limited to the constants *true()* and *false()*. The axioms convey the semantics of the operations. **Spec natural** extends **spec boolean** by adding **sort nat**. The extension allows the declaration of new operators that involve both the new **sort nat** and in general, any sort from the extension which in this case, is limited to **sort bool**. For example,

$$\text{eqnat: nat,nat} \rightarrow \text{bool;}$$

shows that the operators of **spec boolean** may now use both **sort nat** and **sort bool**. The specification is "grown" in this way, a piece at a time.

Parameterized specifications are permitted and are discussed in Yurchak (1984). Their use is minimized, as their properties are not well understood. **Spec string** is the only use of this type of specification.

The semantics and overall structure of the specification must obey certain rules. All symbols must be unique. No symbol may be used unless it has first appeared as the name of a spec, in a sort definition, or to the left of a colon in an operator declaration. This rule guarantees that at no time are the properties of the object inferred by the name ambiguous. Thus, the structure of the specification is much like a Pascal program, but more restrictive. There are no self referential specs, and no use of a spec before it has been defined.

The language classifies all operators into one of three categories, *primitive*, *derived*, and *hidden*. *Primitive* operators are those which must be implemented to provide a full instantiation of the specification. They form the basis of the resource description. As discussed in the previous chapter, not every primitive operation need be directly implemented. However, the full functionality of each primitive operator must be present. **Spec natural** in Appendix B provides an illustration. *Prednat* and *succnat* are primitives that are used to describe the semantics of the arithmetic operators such as *sum* and *sub*. The implementor has the option of excluding either *prednat/succnat* or *sum/sub* because full

_____

discussed in Fasel (1983). We do not use them here.

32

functionality is available from either set of operators. However, if both sets are excluded, then the specification will not be fully instantiated.

Derived operators are those which can be derived from the primitives. The implementor may ignore these operations because it is always the case that their functions can be performed by the composition of primitives. Their inclusion is merely a matter of convenience.

Hidden operators are those to which the programmer has no access. They represent abstractions of the machine required to express a certain semantics. In some cases, they are not necessary but rather a convenience that improves the readability of the specification. Hidden operator **wksrct** (working source rectangle) from **spec pntblktrans** is an example of this kind. In other cases, the hidden operator is essential to the semantic description. An example is the hidden operator **xeq** (execute) from **spec am**. It is not necessary to directly implement hidden operators.

The constructs, *if-then* and *if-then-else* are used to build conditional axioms. The syntax of the conditional axiom is:

> if *boolean expression*
> then
> > *list of axioms*
>
> else
> > *list of axioms*
>
> endif;

As would be expected, the *list of axioms* under the *then* clause, apply only when the *boolean expression* is true. The *else* clause is optional and its *list of axioms* apply when the *boolean expression* is false. The syntax of *boolean expression* is:

> expression meta_relop expression

where the *expression* is any legal expression of the specification. and *meta_relop* is the metalanguage symbol "=" (equality relation) or "!=" (inequality relation). The *truth* of the *boolean expression* is based on metalanguage truth which should not be confused with the specification constant *true()*.

As stated in Yurchak (1984), "the specification does not describe the way in which strings from an alphabet are associated to the elements of the ordinal types such as **spec natural** or **spec integer**, nor would it be appropriate." As a

consequence, we are obligated to refer to the "number" 5 as

$$\text{succint}(\text{succint}(\text{succint}(\text{succint}(\text{succint}(\text{zeroint}())))))$$

If large numbers must be referenced, as in our case, this limitation is impossible. Thus, we have added an enhancement to rectify the problem. The construct $[n]$ allows an arbitrarily large number of operator applications to be expressed. This construct has the following semantics:

$$\begin{aligned}
&\text{if } n = 1\\
&\text{then}\\
&\qquad [n]A(x) = A(x);\\
&\text{else}\\
&\qquad [n]A(x) = [n\text{-}1]A(A(x));
\end{aligned}$$

where n is a positive number, A is an operator, and x is an expression of the same sort as A. With this construct the "number" 100, is expressed as

$$[100]\text{succint}(\text{zeroint}())$$

Note that n and 1 are elements of the *metalanguage* and A and z_i are elements of the specification.

B. THE MACRO PREPROCESSOR

Writing a specification generates voluminous amounts of specification code. In many areas this code is repetitious and can be condensed when a macro preprocessor is available. Besides helping the writer, the macro definitions aid readability. Elements of the specification that share a common macro definition are easily identified. Understanding one implies understanding of all.

The following description of the preprocessor is taken from Yurchak (1984). The basic form of a macro definition is

replace "text..." with "other text..."

Since the grammar of our specification language does not require quotes, they are used as delimiters for the definition and replacement strings. A macro with arguments looks like

replace(A,B,...,Z) "text....." with "other text....."

where the formal parameters *must* be capital letters. Since we do not allow

uppercase letters within the spec itself, an uppercase letter denotes a formal parameter to a macro. Thus for the definition

$$\text{replace(S)}$$
$$\text{"typeof(S);"}$$
$$\text{with}$$
$$\text{"typeS: } \rightarrow \text{ type;}$$
$$\text{atomofS: val } \rightarrow \text{ S;}$$
$$\text{valofS: S } \rightarrow \text{ val;"}$$

then the string

$$\text{typeof(bool);}$$

would be replaced by

$$\text{typebool: } \rightarrow \text{ type;}$$
$$\text{atomofbool: val } \rightarrow \text{ bool;}$$
$$\text{valofbool: bool } \rightarrow \text{ val;}$$

wherever it appeared.

The utility of the macro is demonstrated in the following area which is particularly repetitive. The **fetch** and **store** operators are used to store and retrieve values of any type to and from primary memory. All AM data types map into a common sort, *val*, which is passed to or returned from these storage operators. In **spec typing** there are over thirty data types whose mapping functions must be described. Each set of mapping functions is identical except for the names of the operators and sorts. The macro definitions essentially allow us to describe the first data type and then just list all the others.

Macro definitions are excellent for expressing certain properties of operators such as commutativity, transitivity, etc., which are used throughout the specification. Rather than write out the associated axioms repeatedly, we define macros with appropriate parameters which permit a more readable and explicit expression of these properties. The equality operator for integers example from Yurchak (1984) illustrates.

$$\text{eqint: int,int } \rightarrow \text{ bool;}$$

which returns **true()** if the arguments are equivalent, **false()** if not. We would like to express **eqint** as an equivalence relation on objects of type **int**. Thus, we

need the following axioms:

```
eqint(i,i) = true();
eqint(i,j) = eqint(j,i);
implies(and(eqint(i,j),eqint(j,k)),eqint(i,k)) = true();
```

But there are relations like this one throughout our specification. Thus we define macros like

```
replace(X,S)
        "equivrel(X,S);"
with
        "for i in S
                X(i,i) = true();
        for i,j in S
                X(i,j) = X(j,i);
        for i,j,k in S
                implies(and(X(i,j),X(j,k)),X(i,k)) = true()"
```

which permits us to write, in the case of **eqint**,

$$equivrel(eqint,int);$$

We then read this as "**eqint** is an equivalence relation on **int** ". Note that we are not required to explicitly specify the type of free variables, since this can normally be determined by context. We do so in the interest of clarity. since there can be no doubt for which type **eqint** is an equivalence relation.

For the reader who doubts that the more complex macros described in this specification will work, a modified version of the familiar M4 macro preprocessor[5] will correctly deal with every macro found in our specification. As a matter of convention, all macro definitions are located at the beginning of the specification.

---

[5] See Kernighan and Ritchie, *The M4 Macro Preprocessor*, Bell Laboratories, Murray Hill, New Jersey, July 1974.

V. THE DESIGN

The design of the display resource is an exercise in building an abstraction or model of the display that is appropriate for representing and manipulating images. Our intention is to model a RGB color monitor driven by a bit-mapped display system. Appendix B contains the complete specification for the AM machine and will be referred to throughout this chapter.

This specification most likely contains errors despite our best efforts to be exact and thorough. Incorrectly written axioms that introduce ambiguity are one type of potential error. A second type of problem is the possibility that a portion of the spec is incomplete which would allow legal but undesirable implementations. Finally, we note that while this specification is, in fact, a description of **a resource**, there is no guarrantee that our perception of this resource is precisely what the specification describes.

A. THE IMAGE CONCEPT

Any abstraction may take on various forms. Some views are more useful than others. The key question is just how should a monitor be abstracted. What are the *things* that a programmer wishes to manipulate when he works with a display? The goal is to provide a monitor abstraction that supports the concepts of contemporary graphics systems. The set of functions that control the monitor image should correspond to the operations on a set of objects that represent the natural way of thinking about the task of creating a visual display. This monitor should encourage programmers to work in terms of images that are used to construct more complex images. The visual image is the main entity or object that we manipulate with the display resource. All our efforts are directed towards its construction, modification and projection. In order to effectively manipulate this entity, we wish to describe its properties along with a set of operations that can modify these properties. In short, we want to develop an abstract data type that models the visual image effectively.

Extensive research has been performed in this area during the development of Smalltalk-80, Goldberg (1983). We have found their results extremely helpful and have incorporated many of their concepts into our abstraction. The object oriented approach of Smalltalk is closely related to abstract data types and is very convenient for expressing abstractions. There is a close correspondence between the abstract data type with its set of operators and the Smalltalk object with its set of messages.

The abstract notion of an image, with its properties and operators, is highly developed in Smalltalk-80. The entity *form* represents the *image* in both Smalltalk (as an object) and AM (as an abstract data type). The *form* is a two dimensional object. A number of different sized *forms* may coexist at any point in time. Each has a height and width, and for our purposes is always rectangular shaped. That property which enables us to *see* the form, is a two dimensional array of points, each of which is an individually controlled source of illuminance. The *area* of the form requires the concept of *spatial reference*. Smalltalk-80 has a very general and powerful set of image operators embodied in the *Bitblt* class. These operations allow forms to be combined in whole or part with one another to create a new forms. Most applications of *Bitblt* operations are graphically simple, such as "move this rectangle of pixels from here to there". However, the *Bitblt* class can also be used for some very powerful and sophisticated operations involving transparency, image combination and masking. Smalltalk-80 does not develop the concept of color. Conceptually, color images are only a minor increase in complexity. But conceptual operations such as inverse video and masking take on a completely new look when color is added.

From the programmers point of view, consider how images are created and manipulated. The images that we wish to create are not unique to an electronic screen. The same images may be created on paper, photographed or drawn on any flat or two dimensional surface. When a person views a picture, he usually identifies distinct objects within the picture. For example, consider a picture of a room which contains a man holding a basketball in front of him. Depending on the level of detail, there are three distinct objects that combine to create

the whole picture, the *room,* the *man* and the *ball.* Note that the objects are not involved, only the images that are associated with the objects.

The most straight forward way we might create the above scene, would be to begin by drawing the image of the room on a clean sheet of paper, since it is the object that is farthest back. The image of the man would be drawn next. He is in front of the two dimensional image of the room and therefore covers portions of the previous work. Finally, the image of the ball is drawn. It covers parts of both the man and the room. This is the simplest approach and assumes that there is no difficulty in overwriting the previous markings. An artist using a pencil and paper would first conceptualize how to construct this scene in his minds eye. He considers which objects are in front of others and which details of an object should be hidden. But because these materials do not easily overwrite, he may only draw the final or composite image.

A color monitor does not have the restrictions of a pencil and paper. It is easily overwritten. But this fact has not always been used to full advantage. Often the image on a monitor is created the same way an artist would do it on paper. The composite picture is constructed in the programmers head and the monitor programmed to display this picture. This approach keeps the programmer operating at a very low level of abstraction. The computer is unable to provide assistance except at the lowest levels. The trend in contemporary graphic systems is to take the straight forward approach of overlaying images on top of other images. Once an image has been created, it may be used as a building block for a more complex image. The programmer now works at a much higher level of abstraction. The power of the computer can be unleashed to "construct images" rather than just a series of points and lines.

The process of creating a monitor picture boils down to creating and manipulating forms that may then be fed to the monitor for display. To create the previous scene, one would start with a blank full sized form that fits the display screen. The programmer would then manipulate this form to create the image of the room. Using the same technique, he would create a form for both the man and basketball. He would then use the man form as the source and combine it with the room form as the target. The basketball form is then used as

the source and combined with the resultant form as the target. The final composite form is then fed to the monitor and displayed.

Copying portions of a form would be useful. The programmer may already have the man in another form and wish to reuse it. He should be able to copy it out by combining just that part of the form which contains the man as the source.

How is text handled in such a world? Essentially, the image of any character is nothing more than a form. Adding text to a picture only requires that the corresponding character form be copied onto the target form at the proper spot.

The remainder of the display abstraction consists of details needed to support the form concept. Intensity and color data types provide the coloring concept while points and rectangles provide the spatial references. A character set is just a list of forms or icons. This notion is supported with the font data type which is an indexed array of forms.

The preceding is a topdown view of the image abstraction. In the remaining sections of this chapter, we take a more detailed look at the issues and design of both the data abstractions and resource abstraction.

B. ABSTRACT IMAGE DATA TYPES

In this section, we develop the abstraction of the image in detail along with other data types needed to support it. In addition, we discuss issues pertinent to the design and examine how the specification captures the properties of the abstraction.

If programs are to be portable, hardware details of the target machine must be kept hidden. Consider pixel density. No matter what pixel density we choose for AM, it will differ from many machines. The way to achieve portability is to provide an interface that appears to have a standard pixel density. Any implementation whose hardware differs from this standard, will provide a hidden mapping from the standard to the physical. This general technique is used several times to hide hardware details.

An important design issue is raised when this technique is employed. On what basis do we choose our *standard*, or in the above case, the pixel density? The specification should be written to allow the *programmer's intent* to be

captured. If the *standard* is chosen too small, the capability of powerful hardware can not be realized. If it is too high, it may place an excessive burden on the small machine and is awkward to program. The idea is to provide *granularity* sufficient to record the programmers intent at any *reasonable* level of detail. Implementations built on powerful hardware will display the programmers intention in full detail. Lesser machines will display only the detail within their capacity. In general, we consider *reasonable* to include present and near future capabilities.

1. Intensity and Color

Each point in a form or image has a color attribute which is represented by the *color* data type. **Spec pointcolor** expresses the semantics of color. In an RGB system, the color at a point on the CRT screen is a composite of three primary colors, red, green, and blue. Each primary color is associated with an intensity which is the measure of the illuminance emitted. The color is controlled by varying the intensities of the three primary colors.

The *intensity* data type, **spec intensity** models the intensity attribute of a primary color. It is a scalar quantity which is similar to the naturals except that it has an upper bound. Its elements have an ordering that are bounded with the constants *minintens()* and *maxintens()*

The intensity range is a design choice. In reality, this range is continuous but there is a limit to the number of discrete intensity levels the human eye can detect. Our range should be wide enough to support any level of detail the programmer may *reasonably* want. Foley (1984) indicates that 128 levels of intensity is not quite sufficient to provide smooth contouring, thus 128 is too narrow. Our choice of 200 is somewhat arbitrary. This number is less than 256 which allows an intensity to be represented with 8 bits.[6] We choose 200 rather that 256, because it provides sufficient detail and is easy to convert to a scale of 1 to 100 for relative range comparision, i. e., 160 is 80% of the maximum.

The intensity data type provides counting, addition and subtraction operations like the naturals plus the full set of relational operators. Note that the

---

[6]While we don't wish to dictate how intensity will be represented, implementation considerations are always important.

$$maxintens() = [199]succintens(minintens())$$

defines *maxintens()* in terms of *minintens()*. Neither constant is tied to any natural number. Most implementors will probably represent portions of intensity with a subset of the naturals, but, they are not forced to do so. There are several reasons why intensity should not be synonymous with natural. One, natural has no concept of *null* which is discussed below. Two, intensity has a maximum value and natural does not. Finally, the natural operations of multiply and divide have no meaning to intensity.

How to treat boundary conditions is another design choice. What does the *succintens(maxintens())* mean? One possibility is to let

$$succintens(maxintens()) = maxintens();$$

This prevents an error occurrence which is attractive, but it leads to some unexpected properties. Should intensity addition:

$$150 + 100 = 200 \text{ (the maxintens is 200)}$$

be permitted without warning the programmer? Our view is that this is an error. It does not make sense and we don't really know what the programmer intended. To prevent this, we include axioms:

$$predintens(minintens()) = undef;$$
$$succintens(maxintens()) = undef;$$

Careful examination of the *sumintens* and *subintens* axioms reveal that these operations are also *undef* if their execution would lead to a result that is out of bounds.

Intensity has one disjoint element, the constant *nullintens()*. Its purpose is to support the concept of *transparent color* which will be developed shortly. *Nullintens()* may not be used in any of the intensity operators except *eqintens* and *neintens*. It is meaningless to take the successor of *nullintens()* or add another intensity to it. Likewise, *nullintens()* is neither *greater* nor *less than* any other intensity. It can only be compared to other intensities for equality. For

each of these forbidden operations, there is an axiom that explicitly declares the operation to be *undef* when *nullintens()* is an argument.

Color is a triple of three intensity components. The operator *defcolor* is used to associate the three intensities that define a specific *color*. *Redcompnt, grncompnt,* and *blucompnt* return their respective intensity component of a color. Two colors can be tested for equality by comparing each of its intensity components. *Greater* and *less than* relations do not apply because colors do not have an inherent ordering.[7]

The notion of *transparency* is provided by *nullcolor* and provides the means to control form overlays and copying. When a form is constructed to represent an image, points associated with the object are given the desired color. Points not part of the image are colored *transparent*. This distinction allows points in the *bottom* form to show where the *top* form is *transparent* and be replaced where it is not. The transparent color must be unique to all other colors. If a color, say black, were to be used, it could never be used as the *black color* because it would be impossible to tell if a *black point* is part of the image or a hole. *Nullcolor()* is defined to be that color whose three components are *nullintens()*. It is undefined to construct any color with a mixture of null and non-null intensities.

2. Spatial Reference

In order to talk about the area of an image and the location of individual points, we need a spatial reference system. **Spec point** provides the data class that supports the coordinate system. **Spec rectangle** provides the notion of area.

A point is represented by **sort pnt** and constructed from an ordered pair of **int**'s. The operator *locpnt* is used for this construction and the operators *xcord* and *ycord* return the respective coordinates of the point. Notice that, like the integer data type, the coordinates of a point are unbounded in both the

---

[7]A partial ordering is possible based on a color's luminescence. The intensity of a color displayed on a monochrome screen is the algebraic addition of the luminescence of each intensity component where green is 59%, red is 30%, and blue is 11% of full luminescence. This ordering is only partial because there are many ways to achieve any luminescence level, i. e., 90% of red, green and blue is 90% luminescence, but so is 83% green and 100% red and blue.

positive and negative directions. A full set of relational operators are provided. These operators are important mechanisms that are used to describe other data type operators. The descriptions are possible without the relational point operators, but, their existence make the specification much easier to read. *Gtpnt(a,b)* is true if both coordinates of pnt **a** are greater than pnt **b** 's coordinates, that is, if **a** is above and to the right of **b**. *Gepnt(a,b)* is true if pnt **a** is equal to pnt **b**, *gtpnt(a,b)* is true or if pnt **a** is directly above or directly to the right of pnt **b**. *Ltpnt* and *lepnt* test for the corresponding *less than* relationships.

In the AM resource, increasing the **x** and **y** coordinates moves a point right and up, respectively, which puts the origin of the positive quadrant in the lower left hand corner. Note, however, that the orientation of the coordinate system is not actually defined. Technically, the implementor could orient the coordinate system in any direction with complete correctness, and create chaos if programmers write from a different perspective. This problem sometimes occurs at the interface of an abstraction and the physical world. How do we specify *right* and *up?* We could rename the *gtpnt* operator to *right_and_above_pnt*. But the meaning would still be dependent upon our perception of *right* and *above*. In our estimate, it is more convenient to name these relational operators *greater than* and *less than*. An *English description* must accompany the specification to describe the orientation of the coordinate system.

The last operator provided in the data type is *offsetpnt*. It provides the capability to reference a point by a two part offset relative to another point.

The area enclosed by a rectangle is represented by **sort rct** and is constructed from a pair of **pnt** 's. The operator *area* performs this construction from a point pair which represents either of the opposing corners of the rectangle. Based on our understanding of the coordinate system orientation, the operator *origin* returns the point at the lower left corner of the rectangle and *corner* returns the upper right. The dimensions of a rectangle can be obtained by the *xdimrct* and *ydimrct* operators.

Several other rectangle operators are useful to both the programmer and the specifier. A rectangle may be moved an offset distance, relative to its present

location, with *shiftrct* or moved to an absolute location with *putrct*. *Inrct* is a test operator that checks if a point is inside a given rectangle.

The operator *intsctrct* returns the rectangle formed from the intersection of two rectangles. Describing its semantics with axioms reveal some subtleties. Axiom:

$$
\begin{aligned}
&\text{inrct}(p,\text{intsctrct}(r1,r2) = \text{and}(\\
&\qquad \text{inrct}(p,r1), \\
&\qquad \text{inrct}(p,r2) \\
&);
\end{aligned}
$$

is a straight forward description of when a point is in the intersection, i. e., only if it is in both rectangles. This axiom also handles disjoint rectangles, correctly. But what is the *rct* returned by *intsctrct?* More specific, what is the *origin* of such a rectangle? We must be careful that anomalies of this type do not creep into the specification. This problem is solved by defining the operator *disjrct* which returns *true()* if two rectangles are disjoint. This test operator is then used in the axiom:

$$
\begin{aligned}
&\text{if disjrct}(r1,r2) = \text{true}() \\
&\text{then} \\
&\qquad \text{intsctrct}(r1,r2) = \text{undef}; \\
&\text{else} \\
&\qquad \text{inrct}(p,\text{intsctrct}(r1,r2)) = \text{and}(\\
&\qquad\qquad \text{inrct}(p,r1), \\
&\qquad\qquad \text{inrct}(p,r2) \\
&\qquad); \\
&\text{endif};
\end{aligned}
$$

to specify the undefined rectangle. Note that we can prove from the axioms that the origin and corner are in the rectangle formed from an intersection. Thus. the origin and corner are uniquely characterized but the axioms do not show how they are computed from an intersection.

3. Forms

As we mentioned earlier, the *form* is the data type that represents an image. It consists of a rectangular area and a bit-map which can be thought of as an array of colored points. The height and width of the rectangle impose a two-dimensional ordering on the data in the bit-map. **Spec imageform** specify

45

the properties of this data type. The *initform* operator, with a rectangle argument, creates new forms. *Farea* is used to query the rectangle on which a form is based. Initially, the form's points are all colored transparent. The color of individual points may be read or set with the respective operators, *getcolor* and *setcolor*. The axiom:

$$\text{if inrct(p,farea(f)) = false()}$$
$$\text{then setcolor(p,c,f) = undef;}$$

does not permit the programmer to set the color of a point beyond the bounds of the form. Such an attempt is probably the result of an error. The axiom:

$$\text{if inrct(p,farea(f)) = true()}$$
$$\text{then getcolor(p,setcolor(p,c,f) = c;}$$
$$\text{else getcolor(p,f) = nullcolor();}$$
$$\text{endif;}$$

expresses that when a point is set to a color, the point has that color. Unlike the above first axiom, the *else* clause permits points beyond the boundary to be *read* but specifies that all such point are *nullcolor()*. Reading a point beyond the boundary of a form is not an error. In fact, it is a common occurrence. When the displayed form is smaller than the display window, points outside the form are included which the monitor must read and display. This axiom specifies that that portion of the display window that is outside the form will be displayed in the background color. *Fillform* sets all form points to a given color. This operator is more than just a convenience item, its effect on efficiency is explained in the last section of this chapter.

The notion of *inverse video* is a useful tool to the display programmer. When properly used, it can expand the bandwidth of communication from the computer to the user. It is typically used to draw attention to a portion of the screen or indicate a status such as the current menu selection. One way inversing can be done is to redefine the form in different colors. A second option, is to create and save both instances of the form, and selectively display the desired instance. An inversing operator is a better solution particularly in applications where there are many forms to store or the forms need to be highlighted in different colors.

Inverse video on a monochrome screen, is a simple and common operation. If AM where monochrome, the inverse operator *invform,* which produces the inverse image, would only need one argument, the form to be inversed. The image would be displayed simply by reversing the single foreground color with the background color. Introducing color means that the foreground color may now be more that one color. There is no problem changing multiple foreground colors to the one background color. But which of the foreground colors should be used as the new background? The solution is to provide two color arguments to *invform.* The first color specifies the new color for all foreground colors and the second, the background.

The concept of *inverse video* coupled with *transparency,* give the programmer considerable flexibility. For example, the display screen can be set to any background color. As forms are created, modified and displayed, those portions that are *background* are distinguished by the transparent color. The image can take on the quality of *depth.* If an object is *in front of* another, it covers the back object. The background color of the screen *shows through* if nothing is *in front* of it. At any time, the background color may be independently changed without recomputing the images. Any font can be easily *inversed* to any color pair. For instance, imagine a white and green icon on a screen with a blue background. If it is inversed to white and red, the background portion of the icon which was formerly blue becomes red, and the image which was formerly white and green is now white. Or, it could be inversed to blue and white which creates a white rectangle on a blue field with the blue *bleeding through* the rectangle to form the icon.

In addition to the above operators, we provide a series of operators that combine forms to create other ones. These operations are grouped together in the data type *point block transfer* which is discussed in a separate section.

4. Fonts

Spec iconfont represents the *font* data type which is an indexed array of *forms.* Its most obvious use is the representation of a character font. A *character* is not an image but an abstract data type. We commonly associate a character with a number (usually ASCII) and then use that number to index a

character font to get its image. In our case, each form in the font represents the image of the character associated to its index number. This data type is general and not limited to just character fonts. A set of icons corresponding to a set of menu choices is another instance of its use.

The rectangular size of the forms, that make up the font, may be of any size. The only restriction is that their size must all be the same. *Initfont* establishes the form size for a new font. *Rctfont* queries an existing font for its form rectangle. Forms are added and deleted to the font by the respective operators, *setfont* and *delfont*. The axioms:

```
if infont(id,ft) = true()
then
        lenfont(setfont(id,f,ft)) = lenfont(ft);
else
        lenfont(setfont(id,f,ft)) = succnat(lenfont(ft));
endif;
```

and

```
getfont(id,setfont(id,f1,setfont(id,f2,ft)) ) = f1;
```

show that the *setfont* operator replaces any existing icon and adds the icon if it does not already exist. The axioms:

```
if infont(id,ft) = true()
then
        lenfont(delfont(id,ft)) = prednat(lenfont(ft));
else
        lenfont(delfont(id,ft)) = lenfont(ft);
endif;
```

and

```
infont(id,delfont(id,ft)) = false();
```

express the notion that delfont deletes an icon if it is in the font and does nothing if not. *Lenfont* returns the number of forms in the font and *infont* checks if the font has a form associated with a given index. The operator *getfont* returns the form at the given index. Axioms:

48

$$\text{if infont(id,ft)} = \text{false() then}$$
$$\text{getfont(id,ft)} = \text{undef;}$$

and

```
if and(
        eqint(xdimrct(rctfont(ft)),xdimrct(farea(f)) ),
        eqint(ydimrct(rctfont(ft)),ydimrct(farea(f)) )
) = false()
then
        setfont(id,ft) = undef;
endif;
```

state that it is undefined to get a nonexistent icon or put an icon in a font that is the wrong size.

When a display consists of text, it is convenient to use a coordinate system that is based on the character size rather than a point. We introduce the term *spot* to indicate a given icon position. The position of an icon *spot* is defined as the point at its lower left corner. The point positions of a series of *spots* are in multiples of the font rectangle size. For example, if the font rectangle is 10 points wide, the x coordinates of a row of spots might be 0, 10, 20, 30, etc. The y coordinate would increment in a similar fashion.

Three font operators are provided to map coordinates between the point and spot coordinate systems. Note that the spot coordinate system varies depending on the rctfont size. The operations, *spmap* (spot to point) and *psmap* (point to spot), map spot coordinates to points and back respectively. From the axiom for *spmap:*

```
spmap(p,r) = locpnt(
        mltint(xcord(p).xdimrct(r)),
        mltint(ycord(p),ydimrct(r))
);
```

we see that, given a *rctfont* of 50 by 50, *spmap* takes the spots (0,0), (1,1), and (1,2) to points (0,0), (50,50), and (50,100). The axiom for the inverse operation is:

$$psmap(p,r) = locpnt($$
$$divint(xcord(p),xdimrct(r)),$$
$$divint(ycord(p),ydimrct(r))$$
$$);$$

which returns the series of points back to the original spot coordinates. If the *rctfont* is changed to 10 by 10, these points now map back to spots (0,0), (5,5), and (5,10). We use these operators to access locations in a form, relative to the font size in use. These operators both use point (0,0) as their origin. If a form's origin is not a multiple of the font rectangle from point (0,0), then the spot coordinates will not be evenly aligned with the form's bottom and left boundary. When different font sizes are used simultaneously, each font uses a coordinate system with a different scale. Conflicts are likely to arise and it is the programmers responsibility to keep things straight.

Spots may be referenced using a third operator *offsetfont*. This operator is also based on the font size, but, it is relative to an arbitrary point rather than point (0,0). This operator is convenient for relative screen addressing. Since it is based on the given reference point, it may be used to position icons anywhere in a form. It is also useful when working with several fonts of different sizes.

C. POINT BLOCK TRANSFER

Spec pntblktrans represents the *point block transfer* data type. The main operator is *copyblt* which is used to combine a designated portion of one form with a portion of another form. This operation is complex and requires seven parameters. Three of these parameters specify the forms to be used as the *source*, *destination*, and *mask*. The other four parameters are encapsulated in the sort **ptblt** which is adapted from the Smalltalk object *Bitblt*. **Ptblt** simplifies the **copyblt** call and is efficient, because often a series of operations is invoked, all of which use the same parameters in the one **ptblt**.

The four parameters of **ptblt** are the *source*, *destination*, and *clipping rectangles* and the *copy rule*. A pair of operators is provided to read and set each parameter. For example, *getsrct* and *setsrct* read and set the source rectangle. The *source rectangle* specifies that portion of the source form to be copied. The *destination rectangle* indicates the location on the destination form where the

copied points will be placed. The *clipping rectangle* is similar to the destination rectangle. It limits the area of the destination form that can be affected by a copy operation. This independently controlled rectangle is provided for windowing onto larger scenes. The clipping rectangle is used to ensure that all picture elements are contained within the bounds of the window.

The clipping function is provided in one central place for efficiency. This avoids the need to replicate the function in all application programs, Goldberg (1983). Normally the clipping rectangle is set large and clipping does not take place. Each application program produces its forms using only the destination rectangle. A controlling program may set the clipping rectangle to establish the desired window. All other application programs may continue normal operations using their destination rectangles with the clipping rectangle overriding any copies that fall outside the window.

The *combination rule* determines how the source points will be combined with the destination. The rule may specify that the source overwrite all destination points or only transparent points. The difference is to place the source image *in front of* or *behind* the destination image. Other rules are available which produce a variety of effects.

The combination rule is applied to one pair of points one at a time. Each pair is formed by a *source* point and its *corresponding* destination point. The mapping function that establishes this *correspondence* will be developed shortly. Each invocation of copyblt involves a third form, the *mask*. Its role is to provide a texture pattern to the image. This function is realized by modifying the source point color before it is compared to its corresponding destination point. The details of the point mappings, mask rules and combination rules will be developed in the following sections.

1. The Correspondence Function

The concept of mapping each source point to a particular destination point is quite simple. The precise specification of this idea is tedious, however. To describe the copyblt operator, we introduce the hidden operators, *copypnt*, *getmcolor*, *modpnt*, *copyrecur*, *wksrct*, *wkdrct*, *nextpnt*, and *matchpnt*. We coin the term *correspondence function* to refer to the collective function provided by

the individual operators that pair destination and source points. Copyblt is described recursively. *Copyrecur* controls this recursion by systematically visiting each point in the source rectangle. *Copypnt* embodies the *combination rules*. The *masking rules* are built into the operators, *modpnt* and *getmcolor*.

Copyblt does not use the source and destination rectangles of **ptblt** without qualification. As illustrated in Figure 5.1 (a) and (b), these rectangles may lie partially or wholly outside their corresponding forms, an event which we do not consider an error. The copyblt operation is based the intersection of the **ptblt** rectangles and their corresponding forms. *Wksrct* and *wkdrct* provides the working rectangles for the source and destination, respectively. The axiom:

Source Form   Destination Form   Target Area

(a)   (b)   (c)
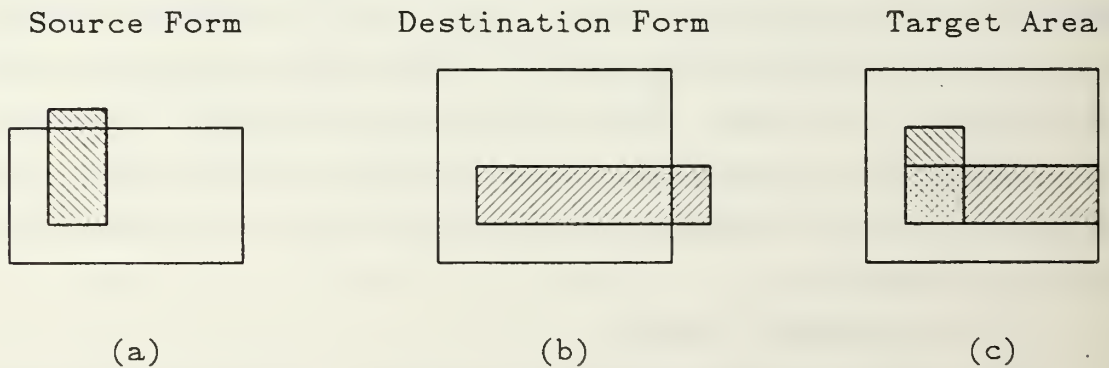
Figure 5.1: Working Rectangles and Target Area

```
        if or(
                disjrct(farea(f1),getsrct(pb)),
                disjrct(farea(f2),getdrct(pb))
        ) = true()
        then
                copyblt(f1,f2,f3,pb) = f2;
        else
                copyblt(f1,f2,f3,pb) =
                        copyrecur(
                                origin(wksrct(f1,pb)),
                                f1,
                                f2,
                                f3,
                                pb
                        );
        endif;
```

expresses that the destination form is unaltered if either the source or destination form is disjoint from its ptblt rectangle. Otherwise, the recursion begins at the origin of the working rectangle. The semantics of *copyrecur* is described in the axiom:

```
        if inrct(p,wksrct(f1,pb)) = true()
        then
                copyrecur(p,f1,f2,f3,pb) = copyrecur(
                        nextpnt(p,f1,pb),
                        f1,
                        copypnt(p,f1,f2,f3,pb),
                        f3,
                        pb
                );
        else
                copyrecur(p,f1,f2,f3,pb) = f2;
        endif;
```

If the point **p** is outside the source working rectangle, then the destination form is unaltered. Otherwise, *copypnt* is called with the current point to modify the destination form. The form it returns will be the destination form for the recursive call at the next point. The *nextpnt* operator systematically visits every point in *wksrct* and then exits out the top of the rectangle, which is the signal to terminate the recursion.

Copypnt is called with the current point in the *wksrct* for each invocation of *copyrecur*. The *matchpnt* operator takes this point and returns a destination point. The *matchpnt* always has the same relative position to the *wkdrct* origin as the input point has to the *wksrct* origin. For instance, if the input point is right two and up four from the origin of the *wksrct*, then the matchpnt will also be right two and up four but relative from the *wkdrct* origin. As shown in Figure 5.1 (c), the effect is to shift the two working rectangles to a common origin which forms the effective intersection labeled **A**. We will refer to this area as the *target* rectangle. Each destination point in the target becomes the matchpnt to the overlapping source point. The points in area **A** are potentially altered by copypnt. The points in area B fall outside *wkdrct* and will not be altered. Points in area C are also unchanged because they do not match any source point. The *clipping rectangle* is built into *copypnt*. Each point sent to copypnt is checked against the intersection of the working destination and clipping rectangles. If it is inside, then the matchpnt is altered according to the mask and combination rules in effect, otherwise, no change is made.

2. Masks

The ability to fill an area with a regular pattern provides the effect of texture. The desired pattern is provided by the third form specified in copyblt. This *mask* form may be any legal form of any size. The mask need only be large enough to represent one cycle of the pattern. The pattern of the mask repeats over the entire target area when the masking rules make it visible.

The *mask rules* are fixed and always in effect. Like the main copy rule, the *mask rule* is based on a comparision of a source point with its *corresponding mask point*. The hidden operator, *getmcolor*, contains the mask rule and returns the resultant color. *Getmcolor* is actually used in the combination rule rather than the source color. The semantics of *getmcolor* are contained in the axiom:

```
If or(
        getcolor(p,f1) = nullcolor(),
        getcolor(modpnt(matchpnt(p,f1,f2,pb),f3) = nullcolor()
) = true()
then
        getmcolor(p,f1,f2,f3,pb) = getcolor(p,f1);
else
        getmcolor(p,f1,f2,f3,pb) = getcolor(modpnt(matchpnt(p,f1,f2,pb),f3);
endif;
```

which, when simplified, says:

```
            if or(
                    source point = nullcolor(),
                    match point = nullcolor()
            ) = true
            then
                    resultant = source color;
            else
                    resultant = mask color;
            endif;
```

Note that the mask may be multicolored. When no texturing is desired, the specified mask form is composed of nullcolor().

   Modpnt is a two dimensional modulo function. It computes the *remainder point* of any point, *modulo* the mask rectangle. This function establishes the correspondence between the points in any given form with the typically smaller mask form.

   We should think of the mask pattern as radiating out from the point (0,0) to create one large pattern that encompasses the whole form. Initially, this pattern is *hidden*. Repeated use of copyblt in the masking mode *uncovers* portions of the formwide pattern. Thus, no matter where the mask pattern is applied, if it were to be extended back to point (0,0) (i.e., the pattern revealed), the origin of the mask will align with point (0,0). This property provides a smooth and uniform texture without disjoint *seams*. The *corresponding* mask point must be based on the *matchpnt* rather than the source point location, if the mask is to have this property. The difference is subtle but has major ramifications. Consider the situation depicted in Figure 5.2. The mask shown in (a), is a checkerboard of black and white points. Assume we are using the
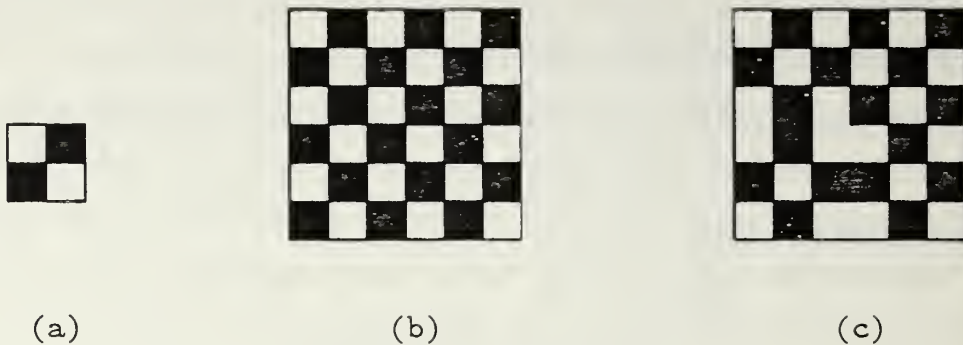
Figure 5.2: Disjoint Seams from the Masking Rule

combination rule which always overwrites the destination with the source, or more precisely, *getmcolor*. Let every point in the form be *even* or *odd* , e. g., points such as (0,0), (1,1), (1,3), and (2,4) are *even* and points (0,1), (1,2), (2,3), and (3,2) are *odd*. If the source form is composed of non-null colors, then the mask will show. No matter how we make a series of copies to the destination, the *odd* points should always appear white and the *evens* black, as in Figure 5.2 (b). But if the mask color is based on whether a source point is *even* or *odd*, disjoint seams can appear. This is because an *even* source point may be mapped to any destination point, *even* or *odd*. Figure 5.2 (c) shows a potential result of incorrectly specifing the mask rule. To eliminate seams, the location of the *matchpnt*, that the source point maps to, is used to choose the mask point. The color of this mask point is then compared to the source point color in *getmcolor*. We see in the *else* clause of the above axiom that *matchpnt* is used rather than **p** directly.

3. Combination Rules

There are eight *combination rules* used by copyblt. These rules and their effect are illustrated in Figure 5.3. The rules are organized in a manner similar to the method used by Smalltalk. Goldberg (1983). Figure 5.3 (a) show a box
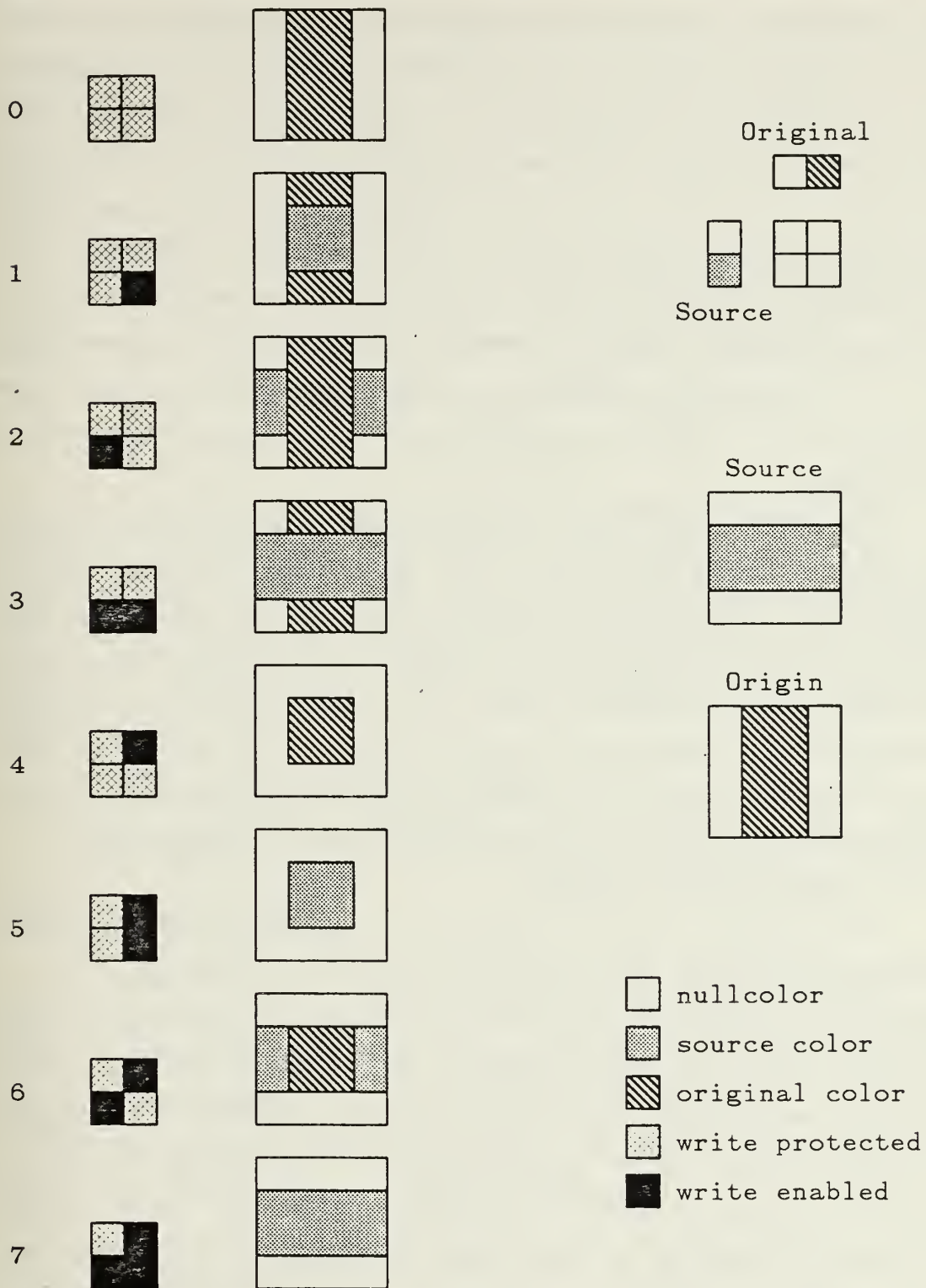
Figure 5.3: Combination Rules

57

with four cells corresponding to the cases encountered when a point from getmcolor is combined with a point from the original destination. For example, *cell 4* corresponds to the case where the getmcolor is nullcolor() and the original destination is a non-null color. By marking each of the cells as *write enabled* or *write protected*, the box can be made to represent any of the combination rules. The numbers of the cells marked *write enabled* add to map to the combination rule number. For instance, *rule 3* means that cells 1 and 2 are write enabled and the remaining cells are write protected. If the getmcolor and destination point pair correspond to a write enabled cell, the destination point is changed to getmcolor, otherwise, it remains unchanged. The upper left unnumbered cell is always write protected. Enabling it would mean that when the original destination point is nullcolor(), it will be replaced with the getmcolor which is also nullcolor().

The effects of rules 0 and 7 are the simplest to describe. Rule 0 prevents any change to the destination and rule 7 replaces the destination with the masked source. One problem with rule seven is that the *background* of the source will leave a *hole* in the destination. This affect is eliminated by rules 2 and 3 which copy only the foreground parts of the source, leaving the destination unchanged, where it collides with the source background. Rule 2 slips the source *under* the original destination object and rule 3 puts the source *on top*. The rules 1 and 6 provide a type of *bleed through* and rules 4 and 5, a *clipping* action.

4. Other Pntblktrans Operations

The operator *drawline* provides its named function by repeated invocations of copyblt. The source form becomes the *brush* shape and the masking and combination rules apply as usual. Lines can be drawn in any width and texture with many different effects. The *drawline* operator is specified recursively using one of two hidden operators, *hdrawloop* and *vdrawloop*, depending on whether the line is more horizontal or vertical. Each invocation of a drawloop operator is relative to a given point on the line. Copyblt is called with the ptblt destination rectangle shifted. Depending on the state of various control variables, the shift is in both the major and minor axes or just in the

major axis. Control variables are updated and the drawloop operator is called again with the next point.

Two additional operators are provided for font operations, *copyfont* and *invcopyfont*. These operators are designed to conveniently transfer icons to a form. *Copyfont* packages the *getfont* and *copyblt* operators together. It places the designated icon in a form at a specified location. *Invcopyfont* is the same as *copyfont* with inverse coloring added.

The *copyblt* operation, is the key to moving the programmer from pixel to image manipulation. The independently controlled combination and masking rules, along with the individual selection of the working rectangles to produce the desired target area, give the programmer great flexibility and power.

D. THE BIT-MAPPED DISPLAY RESOURCE

In the previous sections, we developed the abstractions of the image and described the set of operators that apply to image programming. We will now describe the display resource itself. The *state* of the machine, in AM (version 1.0), is the aggregate of the memory, register, and stack cell contents. In AM (version 2.0), we will extend the *state* to include two new entities, the *display register* and the *monitor*.

1. Background on the Processor Resource

Before we describe the extension to AM, the original processor resource on which this extension is based should be understood. The following is a brief sketch from Yurchak (1984).

In AM (version 1.0), the five primitive data types, boolean, natural, integer, character, and string, form the atomic data types and are referred to as *atoms*. Yurchak discusses the relationship between the data and the conventional machine and the impact of this relationship on portability issues. He identifies the following properties, which are designed into AM to avoid some of the problems that make portability so difficult to achieve.

- In the organization of primary storage, the next logical data item is in the next logical address.
- Except as formally specified, no data type may be accessed, in any way, as another data type.

- Given any arbitrary logical address, the value stored there *and its type* can always be determined.

Hence, AM uses a tagged architecture with some very special characteristics.

The processor portion of AM is an abstraction of a conventional Von Neumann resource with some unconventional properties. The only machine element is called a *value*. All data primitives (atoms) map into values. **Spec typing** describes the relationship of *values* to *atoms*. To illustrate this relationship, consider the multiplication of two integers. We fetch their value representation from two registers, and convert each *value* to its integer *atom* with the *atomofint* operator. These integers are multiplied in accordance with the integer data type specification. The resulting integer is converted back into a *value* with *valofint* for storage into a register.

Primary storage is an array of one or more memory segments, each of which may contain an arbitrary number of cells. Each cell is capable of "containing" any legal data value. Both programs and data may reside together in a single segment. For high speed storage, there are one or more register segments, each of which contain an arbitrary number of registers. AM also has one or more stacks, a heap, and a crude file system. Again, every register and stack cell is capable of containing any type of data.

The basic atomic data types are augmented by several others needed for the execution of programs. These are instructions and memory, register, stack, and file addresses.

2. The Display Registers and Window

The *value* representation of the new data types, intensity, color, point, rectangle, form, font, and ptblt, may be placed in any memory, register, or stack cell. However, the *image* of the form can not be *seen* by the monitor until it is first placed in a *display register*. The concept of the display register is similar to the standard register. There are one or more display register segments, each of which contains an arbitrary number of registers. Display registers are restricted to contain only the value of a form. When forms are used as operands, they may be located anywhere other types of operands are permitted, in addition to the display register.

60

Any form that is loaded into a display register is a candidate for display on the monitor. Associated with each display register is a *display window*. The *display window* area is fixed and the same size for all registers. It represents the portion of the form in the register that will be presented to the monitor screen. The *display window* may be thought of as a *framing mat* which moves over the form, exposing a fixed area of the form to the monitor.

Spec displaywindow represents this notion of the display window. The axioms specify that the dimensions of the window are fixed to a height and width of 10,000 points. The number of points is another design issue. We need a uniform interface so that all programs can assume a *standard* number of points per screen. This number should be large enough to capture the *programmers intent*. If this number is greater than the physical number of pixels for the given hardware, then the implementor will provide a hidden mapping from the abstract points to the physical pixels. The number of addressable pixels in one dimension of the screen is currently greater than 1000 on some machines. 4096 points is probably large enough for the foreseeable future. Our arbitrary choice of 10,000 is based on these facts plus the convenience of working with a power of 10. The axiom:

$$origin(dwin(a)) = atomofpnt(fetchdwin(a,q));$$

establishes the relationship of a variable window location associated with each display register address.

3. The Monitor and Its Attributes

The concept of the *monitor* is very simple from the programmer's perspective. To produce a picture, the monitor reads the display window of the selected display register. The segment of the form in the window may contain the nullcolor. The monitor must map this color to the selected background color for display. The *display window*, associated with the register, is not conceptually part of the monitor. The fact that there may be more to the form than the segment in the display window, is of no importance to the monitor. It only *looks* at the image the display register provides through its window. As mentioned earlier in the form section, when *getcolor* reads a point beyond the bounds of a

form, it returns *nullcolor()*. This requires an "empty" display window to appear as a blank screen in the background color, as would be expected. The *background color* and *selected display register* are considered part of the AM *state* and are the only monitor attributes the application programmer should be allowed to modify. Six other *read only* attributes describe terminal characteristics that the programmer may need, the number of pixels in either axis of the screen, the physical dimensions of the screen, the number of physical intensities available and the number of color planes. These *read only* attributes must be changeable by privileged programs, to allow initial configuration and subsequent hardware changes. **Spec monitor** describes the attributes of the monitor resource.

·4. Portability Issues

The standardized interface of AM is a major step towards portability. However, differences in the physical screen dimensions can not be overcome simply by fixing the number of points per axis to a standard number. If the screen is rectangular, then a 1000 by 1000 "square" will not be displayed as a square. If the "square" is adjusted to appear correct and is then moved to a physical screen with a different *aspect ratio* (AR), it will again appear wrong. It is not possible to put precisely the same picture on two different screens with different AR's. If the picture's AR is maintained, then any scaling factor can at best match the length of one dimension. The other dimension must be either to long or short. If both axes are scaled to fit the screen, then the picture's AR will change and it will appear distorted.

Generally, the programmer wants his picture to fill the available area of the screen with the constraint that none of the picture is clipped and the picture's AR is maintained. In other words, he really wants to program in dimensions relative to the screen dimensions. When we fix the number of abstract points per display screen, dimensions will be relative to the screen size. The picture aspect ratio can be maintained by scaling the dimensions in a generalized way. The screen AR is computed from the screen dimension attributes of the monitor and placed in a variable as a scaling factor. If this factor is used to adjust the coordinates properly, a picture can be created that will respond, as described above, to changes in the screen size and shape. The

screen size attributes may also be used by the programmer who wants to build a shape that remains an absolute size, regardless of the size and shape of the monitor screen, assuming it is physically large enough.

The mapping from the 100 million abstract points (10,000 by 10,000) to a smaller number of physical pixels raises the potential for considerable inefficiency. Consider this code segment to fill a rectangle:

```
for i = 1 to 1000 do
     for j = 1 to 1000 do
             setcolor(locpnt(i,j),c,f);
```

If the physical screen is 500 by 500 pixels, then each physical pixel represents 400 abstract points. This means that, of the 1 million iterations of the above code, only 2500 pixels are actually set. The other 998,500 iterations set the same pixels over and over. The fillform operator is provided to avoid this problem. Filling the same rectangle with

```
initform(area(locpnt(1,1),locpnt(1000,1000)) );
fillform(c,f);
```

will invoke one setcolor instruction for each of the 2500 physical pixels in the rectangle. The redundant iterations are eliminated. This happens because the implementation will automatically scale the point data type values. Point (1,1) remains the same but (1000,1000) becomes (50,50). The essential difference between these two code segments is that the iteration control mechanism of the first code segment is outside the point mapping function embedded in the implementation.

Use of the form operators such as fillform, copyblt, drawline, etc. avoids this type of inefficiency, assuming the implementation is done properly. The iteration in these operations is accomplished after abstract points are converted to real pixels. In most applications, use of setcolor can be avoided after the first several forms are constructed. Once several *brush* and mask forms are created, they can be used with the other operators to construct the forms for display. If the programmer must use setcolor extensively, then he has the option of querying the monitor attributes to get the actual number of pixels. He can then use this information to build some step functions to avoid redundancy.

The intensity and color capability attributes may be useful in a way similar to the pixel information. If an application is designed to run on hardware with a wide range in the number of discrete intensities available, then for powerful machines, the program may devote considerable effort to compute points that differ slightly in color shade. In such cases, the actual intensity capability of the current hardware might be used to scale back the number of computations to avoid generating detail that the implementation will only discard because of hardware limitations.

VI. IMPLEMENTATION

AM is implemented as a finite state machine interpreter. Version 1.0 comprises approximately 12,000 lines of C code, including the assembler. Details of the assembler are treated in Appendix C. The overall concept is quite simple. A text file representing an assembly language program is translated by the assembler into a relocatable object module. A loader, part of the AM interpreter, loads this object module into the appropriate cells, and AM executes it.

The original AM, version 1.0 - Unix, was implemented by John Yurchak on a VAX 780 running Unix 4.2, B. S. D. The Unix version of AM was rehosted to a Zenith Z100 microcomputer running MS-DOS 2.13. Rehosting AM (version 1.0 - Z100) involved all the traditional problems and difficulties of porting software. For example, Unix C allows 12 character names but the Lattice C compiler used on the Z100 only recognizes the first 8 characters. Worse, the MS-DOS linker only uses the first 6 characters. Approximately 350 functions had to be renamed under a more compact naming convention. This involved changes throughout the 12,000 lines of code. A more serious problem involved the passing of structures. The Unix version takes advantage of a non-portable feature of Unix C, the ability to pass structures. For simplicity and safety, we would have preferred to pass structures by value in the Z100 version, as well. However, this was not an option and we were forced to convert the program to pass structures by pointer, instead. To compensate, we adopted a convention that essentially maintains the pass by value properties of the Unix version.

The ordeal of rehosting AM gives one a good dose of the "portability blues". But the effort appears to be worth while. The battery of test programs used to test the Unix version, also runs on the Z100 version without modification and with the same results.

AM (version 2.0 - Z100) is incomplete at this time. The assembler has been revised to handle the full extension including all new data types, the resource extension, plus some additional operators for the original data types. The machine itself has not yet been extended to handle the new data types. However,

we have done sufficient work in this area to give us a very good idea of what the implementation will look like when it is completed.

Yurchak (1984), Chapter 5, provides an excellent description of the AM implementation. For completeness and reader convenience, we will repeat major portions of Yurchak's description, interspersed with examples and discussion of the Version 2.0 extension.

The overview of the implementation is broken into four main areas. These are the representation of data types, the mapping of operators in the specification to functions in the interpreter, the handling of errors, and the execution of a program. Following the overview, we discuss several issues pertinent to the display implementation.

A. IMPLEMENTING DATA TYPES

AM is a tagged architecture. Each data element or value must be self descriptive. It is important to realize the distinction between an atom (data type) and a value. An *atom*, such as a natural, point or form, represents a problem solving abstraction. A *value* is a machine element and is the specific representation of an atom. An atom is representation independent. We establish representation independence with conversion functions that map atoms to values and values to atoms. All abstract atoms map to a specific representation or

---

```
#define T_NAT      0x0002

typedef unsigned intnat;

typedef struct {
        short   type:
        nat     val:
} NAT;
```

Figure 6.1: Type Definitions for Natural

---

value. We say that the machine has a tagged architecture when the atom , that a value represents, can be determined solely from the value itself. The most likely construct to provide this is a structure (record).

Each atom is represented in 'c' as a structure consisting of a 16-bit tag field, and a value field. A sixteen bit code is assigned to each sort in the specification. Whenever an atom is created, or copied, it is tagged with the appropriate code. The size and structure of the value field varies with the type. Figure 6.1 lists some fragments from the header files used to represent the *natural* data type which has a simple value field. Data types with more complex value fields will be described shortly.

·By using a fixed size tag field as the first field in each record, we build in some additional robustness since, even in the event of a mistyped structure being copied into the formal parameter of a function, we can rely upon the first word to be a valid code (the type).

The next step is to describe the structure for machine values, which must be capable of containing any atom. The union structure is the natural choice. Every structure that represents a sort is included in the union, thus, the value structure can represent any atom.

Since the size of the union structure is determined by the largest union member, it behooves us to be careful how we represent each data type. If the value structure becomes large compared to the average size of the member structures, then memory is poorly utilized. The value field of each data type structure either represents the value of the date type or a pointer to its real location. String and form structures are examples that use pointers since their size is variable and usually large. The minimum size for a value structure on the Z100 is six bytes, two for the tag field and four for a pointer. We directly represent a data type's value in the value field if it can be represented in four bytes or less, otherwise a pointer is used. Naturals, integers and intensity are examples.

Figure 6.2 shows the union structure for machine values named *VAL*. Notice that *INSTR* points to a *VAL* and is itself a *VAL* since we must be able to store

```
typedef short opcode;

typedef struct {
        short   type;
        union value   *val;
} INSTR;

typedef union value {
        short           type;
        opcode          opcdval;
        BOOL            boolval;        /* data types */
        INT             intval;
        ...             ...
        ...             ...
        BLT             bltval;
        FONT            fontval;
        MAD             madval;         /* memory address */
        RAD             radval;         /* register address */
        ...             ...
        ...             ...
        INSTR           instrval;
        MOP             mopval;         /* monadic operator */
        DOP             dopval;         /* dyadic operator */
} VAL;
```

Figure 6.2: Machine Values

and fetch instructions. The VAL it points to, is the instruction's opcode. The opcode is actually an array of VAL, consisting of the opcode and operands.

The primary physical resources are also defined as structures. Figure 6.3 illustrates several of these resources. Registers, display registers, primary storage and stacks are represented as arrays of arrays of pointers to values. The reader should note that a simple change to the constants in the header files can completely alter the configuration of the machine. We can specify an arbitrary number of arbitrarily long memory, register, and display register segments, and an arbitrary number of different sized stacks. The monitor is represented as a

```
typedef struct {
     int    size;
     VAL  **val;
} memseg;

typedef struct {
     int    size;
     int    sp;
     VAL  **val;
} stkseg;

typedef struct {
     int    size;
     int    xdwin;
     int    ydwin;
     VAL  **val;
} dregseg;

typedef struc {
     int    xpix;
     ...    ...
     ...    ...
     COLR bg;
     DAD  dsel;
} mtrseg;

#define _NUMMEMSEG   1024
#define _NUMSTKSEG     1
#define _NUMDREGSEG  1

memseg      _mem[_NUMMEMSEG] = {
     1024, 0,
     1024, 0 };
stkseg _stk[_NUMSTKSEG] = {
     512,512,0 };
dregseg      _dreg[_NUMDREGSEG] = {
     32, 0 };
```

Figure 6.3: The Physical Resource

structure of values which contain its attributes. Files are represented as an array of structures containing status information and an input/output buffer. The number and type of files can also be changed by modifying a few constants. Only one module of our interpreter need be recompiled to make this alteration.

B. MAPPING OPERATORS TO FUNCTIONS

It seems natural, although incorrect, to look at the operators in a spec as functions. However, in the implementation, this makes perfect sense. Figure 6.4 lists the code for the AM module which implements the boolean type. The header files which provide the constant definitions are omitted here. Notice that, where possible, we rely upon the operations provided by the C language, rather than slow down an already slow interpreter with axiomatic implementations of the operators.

As the implementation proceeds to more complex specifications, the program relies less upon C and more upon the operators which we have defined. In fact, the more complex operators are implemented as calls to previously defined functions which almost directly mimic the axioms from which they are derived. We will illustrate this shortly.

C. ERROR HANDLING

All errors in the specification are defined with the *undef* operator. By definition, that makes all errors fatal. In general, they need not be. However those errors which are not must be defined explicitly in the specification. As we have said, a more detailed treatment of errors would be an area for further study.

AM flags most errors in the operators which perform data conversions. This is a natural place for this to occur, since it is difficult to see how the type of a data element may be changed at any other time. Figure 6.5 lists a fragment which implements the *natural* conversion routines. The routine *error* does not return, but terminates execution after writing the error message to stderr. Notice that, even if a much larger structure was passed to *atomofbool* or *valofbool,* the error would be detected and handled gracefully. This type of error checking is also performed in the functions which implement data operations.

70

```
BOOL true = { T_BOOL, 1 };
BOOL false = { T_BOOL, 0 };

BOOL *not(a)
BOOL *a;
{
        BOOL *tmp;

        tmp = (BOOL *) tmalloc(sizeof(BOOL));
        tmp->type = T_BOOL;
        tmp->val = !a->val;
        return(tmp);
}

BOOL *and(a,b)
BOOL *a,*b;
{
        BOOL *tmp;

        tmp = (BOOL *) tmalloc(sizeof(BOOL));
        tmp->type = T_BOOL;
        tmp->val = (a->val && b->val);
        return(tmp);
}

BOOL *eqbool(a,b)
BOOL *a,*b;
{
        BOOL *tmp;

        tmp = (BOOL *) tmalloc(sizeof(BOOL));
        tmp->type = T_BOOL;
        tmp->val = (a->val == b->val);
        return(tmp);
}
```

Figure 6.4: Operator-Function Mappings

```
NAT  *atnat(v)
VAL  *v;
{
        NAT  *b;

        if (v->type != V_NAT)
                error("value not of type NAT - %x",v->type);
        b = (NAT *) tmalloc(sizeof(NAT));
        b->type = T_NAT;
        b->val = v->natval.val;
        return(b);
}

VAL  *vlnat(b)
NAT  *b;
{
        VAL  *v;

        if (b->type != T_NAT)
                error("atom not of type NAT - %x",b->type);
        v = (VAL *) tmalloc(sizeof(VAL));
        v->natval.type = V_NAT;
        v->natval.val = b->val;
        return(v);
}
```

Figure 6.5: Error Handling

D. EXECUTION

The final point of interest involves actually executing a program. The method is also illustrative of the way in which the program mimics the axioms of the specification. Here, too, we resort to subterfuge to implement in a finite way a specification which could require the expenditure of an infinite resource (an implied stack in this case). The problem is the corecursive relationship between the functions *req* and *prog*. We eliminate this problem by never actually returning from *req*. We rely on a dangerous but effective C idiom, *setjmp* and *longjmp*. Figure 6.6 illustrates.

```c
#include <setjmp.h>

jmp_buf        _context;

MAD  cond();

main(argc,argv)
char    *argv[];
{
        int     ap;

        for (ap=1; ap < argc; ap++) {
                if (*argv[ap] == '-') {
                        if (*(argv[ap]+1 == 'x') {
                                traceflag = 1;
                                xtraceflag = 1;
                        }
                        if (*(argv[ap]+1) == 't')
                                traceflag = 1;
                }
        }.
        initam();
        amload();
        setjmp(_context);
        Q = prog(&_pc,Q);
        exit(0);
}


STATE        prog(m,q)
MAD *m;
STATE        q;
{
        q = xeq(atinstr(fetchm(m,q)),m,q);
}


STATE        xeq(i,m,q)
INSTR        *i;
MAD *m;
STATE        q;
{
        opnd   *p;
```

```
        if (i->type != T_INSTR)
                error("attempt to execute non-instruction - %x",i->type);
        p = i->val;
        switch (getopcode(p[0].opcodeval)) {
        /*
                a case and semantics for each valid opcode
                goes in here
        */
        default:
                error("attempt to execute an illegal instruction - %x",
                        p[0].opcodeval);
        }
        longjmp(_context,1);
}


MAD  cond(b,m1,m2)
VAL  *b;
MEMADDR *m1,*m2;
{
        return(b->boolval.val? m1: m2);
}
```

Figure 6.6: Program Execution

```
case IM_M_M_:
        q = storem(
                fetchm(
                        &p[1].madval,
                        q
                ),
                &p[2].madval,
                q
        );
        _pc.val = nxtmad(m)->val;
        break
```

Figure 6.7: The Semantics for mov_m_m

In *main*, *initam* configures AM and invokes all of the initialization operators. *Amload* loads a program from secondary storage into the appropriate cells as directed by the linker directives in the object module. *Setjmp* then saves the state of the "real" machine. The variable _pc is the program counter which is set inside *amload*. Now everything is set. The program is loaded and ready to run.

Prog is now called. Notice that prog simply invokes *xeq*. Recall now the axiom which defines the semantics of execution.

$$\text{prog}(m,q) = \text{xeq}(\text{atinstr}(\text{fetchm}(m,q)),m,q);$$

Within *xeq* a large case statement decodes the instruction and executes it according to the semantics provided for that case. This semantics is very closely modeled on the axioms in the specification. Figure 6.7 lists one such case and its accompanying semantic action. Compare it to the axiom for **mov_m_m**.

```
xeq(mov_m_m(m1,m2),m,q) =
    prog(
            nxtmad(m),
            storem(
                    fetchm(m1,q),
                    m2,
                    q
            )
    );
```

The similarities are not accidental. This should make the point that it is beneficial for the implementation language to permit such a close modeling of the specification. Obviously, this made the implementation easier to write, easier to debug and easier to understand.

E. DISPLAY IMPLEMENTATION ISSUES

Implementation of the display portions of the specification present some unique challenges. Limited memory and hardware design place tight constraints on internal representations. To begin with, we can not deal directly with the intensity and point abstractions inside the machine. There are 100 million abstract points in one full screen image, with each point representing one of potentially 8 million colors. Obviously, no machine today can handle this kind of detail for each screen of information.

In Chapter 5, we said that the implementor is responsible for providing a hidden mapping from abstract units to physical units. The question is, when should it be done? The earliest possible time is in the assembler. The assembler can read the abstract information and immediately convert it to physical data. Since the object file would be in terms of physical units, the AM machine would not have to deal with abstract units at all.

The strategy has several major disadvantages. Intensity capability and the number of physical pixels must be known to the assembler to produce the correct object code. Hardwiring this information into the assembler dooms every assembler to serving only one machine. If a new terminal is added, then the assembler would have to be modified. Passing it to the assembler as a parameter is very burdonsome. Requiring the user to intervene with specific hardware parameters defeats one of our major goals. If the assembler is assumed to run on

its target machine, then it can automatically compensate for its current host by reading the monitor attributes. This may be acceptable, but we prefer an assembler to produce object code free of terminal hardward dependencies. Such code would be portable between different AM implementations with a simple code translator based on only the implementation and not on what type of terminal is connected to the system. object code that is not dependent to the terminal hardware and is portable to all AM machines.

For portable object code, the assembler must put the abstract point and intensity information in the output code. This is the strategy we adopt. This information enters and exits AM via the file system, specifically the read and write operators. In the AM system, file data is also tagged. We design the *read* and *write* operators to apply the appropriate mapping to the intensity and point data based on the monitor attributes of intensity capability and number of pixels. For example, the Z100 only has two intensity levels. Therefore, the read operator maps abstract intensities 0 thru 99 to 0 and 100 thru 199 to 1. Conversely, the write operator maps 0 to 0 and 1 to 199. Note the unavoidable loss of information when using a resource with such course granularity.

We now consider some form representation issues. Figure 6.8 shows some of the structures used to represent a form. Like other data types, the *FORM* · structure has a tag which identifies it. The value field points to a *form* structure which serves as an information header. It contains the rectangle of the form and a pointer to the cmap. The *cmap* is an array of colors. The mapping of cmap color elements to specific points is known by virtue of the rectangle origin and its width. Each color element requires a nibble to represent one of nine colors (8 real colors plus nullcolor).

Up to this point, forms are handled much like any other data type. To understand the monitor and display register resources requires knowledge of specific hardware arrangements. The description of the implementation is based on the Z100 hardware. We will attempt to keep this discussion as general as possible.

```
#define T_FORM    0x000a

typedef struct {
        rct     farea;
        char    *cmap;/* colormap */
} form;

typedef struct {
        short   type;
        form    *val;
} FORM;
```

Figure 6.8: Type Definitions for Form

The bit-map for the monitor is located in a reserved portion of memory and is call the *VRAM*. It is organized in three 64k segments, one for each color plane. Essentially, this area is large enough to hold a little over two screenfulls of information. Based on this size, we have the option of creating one or two abstract display registers.

The abstract form is unbounded and any specific form may certainly be larger than the VRAM allotted to a display register. Therefore, when a form is loaded into a display register, a copy must be maintained in regular memory for either that portion of the form that does not fit into VRAM or the whole form. Which approach to use is a trade off between memory and complexity considerations. The loading mechanism which moves a form between the VRAM and the cmap is really a transformation function that converts the form between its two different representation schemes.

The information in the VRAM (and in our abstract display register) is greater than the monitor screen can show at one time. The display window controls what portion of the form is viewed. That portion of the VRAM displayed by the monitor screen is based on a refresh register which marks a start point in VRAM. The monitor refresh cycle begins reading VRAM at this point

and continues reading a fixed number of bytes equivalent to one full screen. We use this mechanism to implement the display window in an efficient way.

When the display register is loaded with a larger form, that part that is actually move to VRAM is based on the current setting of the abstract window. We also fill the excess VRAM assigned to the display window with the adjacent portions of the form. When the window is moved over a different portion of the form, only the refresh register is changed to paint the new picture. This assumes newly revealed portion of the form is already loaded in VRAM. If the window movement is too far and now displays part of the form that is not in the VRAM, then a display interrupt must be executed and the display register is reloaded. Based on which boundary faulted, the portion of the form that is loaded can be biased in anticipation of future window movement. Scrolling text is a common task that will generate this type of action.

The final issue we discuss involves the monitor attributes of background color and display register selection. Conceptually, the background color is a monitor attribute. Its setting affects the image projected from each display register. Changing this setting should immediately change the image in all registers. In reality, the nullcolor is converted to the current background color when the form is loaded into the register because there is no physical definition of nullcolor. Therefore, when the background color of the monitor is changed all registers will be force to reload using the new background color. The display register selection is easily implemented by placing the display window definition of the selected register in the refresh register.

VII. CONCLUSIONS

The investigation and formulation of an abstraction for the bit-mapped display resource was the first of our dual goals. We have developed an abstraction of an image along with supporting data types. We propose that future systems provide a functional set of operators that will encourage programmers to program colored images at a high conceptual level. It is not necessary nor desirable to treat the display as a separate I/O device. Our framework provides an example of a display resource that is fully integrated with the processor, in the same way that arithmetic coprocessors are already used.

Our second goal was to further demonstrate the utility of the axiom specification method by precisely describing the display resource. We believe the processor and display resource are as hard to abstract and formally describe as any portion of a computing system. Recall the copyblt operator. It is an extremely complex operation to formally specify, yet while tedious, the axioms fully describe the operation and are precise, unambigous, and reasonably readable by any experienced programmer. The successful specification of both these resources demonstrates that this method has good merit.

It may turn out that the AM machine is too slow and inefficient to be viable as a practical machine in its present form. We pay a price when we implement in a formal way and with a large number of function calls. How much AM can be modified for efficiency and still retain confidence that it meets the specification is unclear. The difficultly is, at present, we rely heavily on the fact that the code is very similar to the specification. Modifications for efficiency that the make the implementing code less like the specification, erodes our confidence. Improved techniques for testing implementations are needed.

Testing and formal implementation problems aside, the very existence of a precise specification can only improve the portability situation.

- The resource designer and implementor will have a precise target.
- Software designers will have a precise and visible interface on which to build. The semantics of their host will be better understood.

- Quantitative statements can be made about machines that are reportably compatible to a given specification.

abstraction:
          (abstraction spec)?

spec:
          (spechead | parmhead) specbody specend

spechead:
          nameblk 'is'

parmhead:
          nameblk 'parm' specbody 'is'

specend:
          'end' specname ';'

nameblk:
          'spec' specname

specbody:
          extension? specblk

extension:
          extendblk specblk 'end' 'extend' ';'

extendblk:
          'extend' specnames 'with'

specnames:
          specname
        | specnames ',' specname

specblk:
          useblk
        | sortblk? opblk axiomblk?

useblk:
          'use' specname '(' specname ')' mapping? specblk 'enduse'

mapping:
          'where' eqivlist

equivlist:
          equivalence ';'
        | equivlist equivalence ';'

```
equivalence:
        sortname 'is' sortname
      | opname 'is' opname

sortblk:
        'sort' sortnames

sortnames:
        sortname ';'
      | sortnames sortname ';'

opblk:
        primblk? dervblk? hiddenblk?

primblk:
        'primitive' 'op' ops

ops:
        op ';'
      | ops op ';'

op:
        opname ':' arglist? '->' sortname

arglist:
        sortname
      | arglist ',' sortname

dervblk:
        dervops dervdef

dervops:
        'derived' 'op' ops

dervdef:
        'derived' 'def' axioms

hiddenblk:
        'hidden' 'op' ops

axiomblk:
        'axiom' axioms

axioms:
        axiom ';'
      | axioms axiom ';'
```

83

```
axiom:
        conditional
      | ('for' varlist 'in' sortname )? termexpr '=' termexpr

termexpr:
        factor
      | multiplier? opname '(' factors ')'

factors:
        factor
      | factors ',' factor

factor:
        multiplier? opname '(' ')'
      | freevar

varlist:
        freevar
      | varlist ',' freevar

multiplier:
        '[' positive_number ']'

conditional:
        'if' termexpr meta_relop termexpr then else? 'endif'

meta_relop:
        '='
      | '!='

then:
        'then' axioms

else:
        'else' axioms
```

```
            replace()
                "NUMINTENS"
            with
                "199"


replace()
    "DISPLAYSIZE"
with
    "9999"


replace(X,S)
    "equivrel(X,S);"
with
    "X(i,i) = true();
    X(i,j) = X(j,i);
    implies(and(X(i,j),X(j,k)),X(i,k)) = true();"


replace(X,S)
    "reflexive(X,S);"
with
    "X(i,i) = true();"


replace(X,S)
    "commutative(X,S);"
with
    "X(i,j) = X(j,i);"


replace(X,S)
    "transitive(X,S);"
with
    "implies(and(X(i,j),X(j,k)),X(i,k)) = true();"


replace(X,S)
    "associative(X,S);"
with
    "X(i,X(j,k)) = X(X(i,j),k);"


replace(X,S)
    "irreflexive(X,S);"
with
    "X(i,i) = false();"


replace(X,S)
    "symmetric(X,S);"
with
    "implies(X(i,j),X(j,i)) = true();"
```

```
replace(X,S)
    "antisymmetric(X,S);"
with
    "implies(and(X(i,j),X(j,i)),(i == j)) = true();"


replace(S,T)
    "idopers(S,T);"
with
    "startT: → S;
    nextT: S → S;
    prevT: S → S;
    eqS: S,S → bool;"


replace(S,T)
    "idaxioms(S,T);"
with
    "prevS(startT()) = undef;
    prevS(nextS(i)) = i;
    if i != startT() then
        nextS(prevS(i)) = i;
    endif;
    equivrel(eqS,S);"


replace(S)
    "typingopers(S);"
with
    "typeS: → type;
    atomofS: val → S;
    valofS: S → val;"


replace(S)
    "typingaxioms(S);"
with
    "whattype(valofS(t)) = typeS();
    atomofS(valofS(t)) = t;
    if whattype(v) = typeS()
        then valofS(atomofS(v)) = v;
        else atomofS(v) = undef;
    endif;"


replace(S,T)
    "relop(S,T);"
with
    "applyrop(ST(),v1,v2) = valofbool(TS(atomofS(v1),atomofS(v2)) );"


replace(S)
    "isops(S);"
with
    "if whattype(v) = typeS()
        then applybop(isS(),v) = valofbool(true());
        else applybop(isS(),v) = valofbool(false());
    endif;"
```

86

```
replace(S,T)
    "stateaxioms(S,T);"
with
    "fetchS(a,initam()) = undef;
    storeS(fetchS(a,q),a,q) = q;
    implies(
        eqT(a1,a2),
        fetchS(a1,storeS(v,a2,q)) = v
    ) = true();
    implies(
        not(eqT(a1,a2)),
        fetchS(a1,storeS(v,a2,q)) = fetch(a1,q)
    ) = true();"


spec boolean
is
    sort
        bool;
    primitive
    op
        true: → bool;
        false: → bool;
        not: bool → bool;
        and: bool,bool → bool;
    derived
    op
        or: bool,bool → bool;
        implies: bool,bool → bool;
    derived
    def
        or(b1,b2) = not(and(not(b1),not(b2)) );
        implies(b1,b2) = not(and(b1,not(b2)) );
    axiom
        false() = not(true());
        not(not(b)) = b;
        and(true(),b) = b;
        and(false(),b) = false();
        commutative(and,bool);
end boolean;
```

```
spec natural
is
    extend
        boolean
    with
        sort
            nat;
        primitive
        op
            zeronat: → nat;                          /* zero */
            prednat: nat → nat;                      /* predecessor */
            succnat: nat → nat;                      /* successor */
            sumnat: nat,nat → nat;                   /* addition */
            subnat: nat,nat → nat;                   /* subtraction */
            mltnat: nat,nat → nat;                   /* multiplication */
            divnat: nat,nat → nat;                   /* division */
            eqnat: nat,nat → bool;                   /* equal */
            gtnat: nat,nat → bool;                   /* greater than */
        derived"
        op
            ltnat: nat,nat → bool;                   /* less than */
            genat: nat,nat → bool;                   /* greater or equal */
            lenat: nat,nat → bool;                   /* less or equal */
            nenat: nat,nat → bool;                   /* not equal */
        derived"
        def
            ltnat(n,m) = not(or(gtnat(n,m),eqnat(n,m)) );
            genat(n,m) = not(ltnat(n,m));
            lenat(n,m) = not(gtnat(n,m));
            nenat(n,m) = not(eqnat(n,m));
        axiom
            prednat(zeronat()) = undef;
            prednat(succnat(n)) = n;
            succnat(prednat(n)) = n;
            sumnat(n,zeronat()) = n;
            sumnat(n,succnat(m)) = succnat(sumnat(n,m));
            subnat(n,zeronat()) = n;
            if gtnat(n,m) = true()
            then
                subnat(n,succnat(m)) = prednat(subnat(n,m));
            else
                subnat(n,succnat(m)) = undef;
            endif;
            mltnat(x,zeronat()) = zeronat();
            mltnat(x,succnat(zeronat()) ) = x;
            mltnat(x,y) = sumnat(x,mltnat(x,prednat(y)) );
            if y = zeronat()
            then
                divnat(x,y) = undef;
            else if ltnat(x,y) = true()
            then
                divnat(x,y) = zeronat();
            else
                divnat(x,y) = sumnat(
                    succnat(zeronat()),
                    divnat(subnat(x,y),y)
                );
            endif;
            endif;
            eqnat(n,m) = eqnat(succnat(n),succnat(m));
            gtnat(succnat(n),n) = true();
```

88

```
            equivrel(eqnat,nat);
            irreflexive(gtnat,nat);
            irreflexive(ltnat,nat);
            transitive(gtnat,nat);
            transitive(ltnat,nat);
            transitive(genat,nat);
            transitive(lenat,nat);
            antisymmetric(genat,nat);
            antisymmetric(lenat,nat);
            symmetric(nenat,nat);
            commutative(sumnat,nat);
            commutative(mltnat,nat);
            associative(sumnat,nat);
            associative(mltnat,nat);
      end extend;
end natural;
```

```
spec integer
is
    extend
        boolean,
        nat
    with
        sort
            int;
        primitive
        op
            zeroint: → int;
            ntoi: nat → int;                                    /* nat to int */
            iton: int → nat;                                    /* int to nat */
            absint: int → int;                                  /* absolute value */
            predint: int → int;
            succint: int → int;
            sumint: int,int → int;
            subint: int,int → int;
            mltint: int,int → int;
            divint: int,int → int;
            modint: int,int → int;                              /* modulo */
            eqint: int,int → bool;
            gtint: int,int → bool;
        derived
        op
            ltint: int,int → bool;
            geint: int,int → bool;
            leint: int,int → bool;
            neint: int,int → bool;
        derived
        def
            ltint(n,m) = not(or(gtint(n,m),eqint(n,m)) );
            geint(n,m) = not(ltint(n,m));
            leint(n,m) = not(gtint(n,m));
            neint(n,m) = not(eqint(n,m));
        axiom
            predint(succint(n)) = n;
            succint(predint(x)) = x;
            ntoi(zeronat()) = zeroint();
            ntoi(succnat(n)) = sumint(succint(zeroint()),ntoi(n));
            iton(zeroint()) = zeronat();
            if ltint(x,zeroint()) = true()
            then
                iton(x) = undef;
            else
                iton(succint(x)) = sumnat(succnat(zeronat),iton(x));
            endif;
            if ltint(x,zeroint()) = true()
            then
                absint(x) = subint(zeroint(),x);
            else
                absint(x) = x;
            endif;
            sumint(n,zeroint()) = n;
            sumint(n,succint(m)) = succint(sumint(n,m));
            subint(x,zeronat()) = x;
            subint(x,succnat(y)) = predint(subint(x,y));
            mltint(x,zeroint()) = zeroint();
            mltint(x,succint(zeroint()) ) = x;
            mltint(x,y) = sumint(x,mltint(x,predint(y)) );
            if y = zeroint()
```

90

```
                then
                    divint(x,y) = undef;
                else if ltint(absint(x),absint(y)) = true()
                then
                    divint(x,y) = zeroint();
                else if or(
                    and(
                        gtint(x,zeroint()),
                        gtint(y,zeroint())
                    ),
                    and(
                        ltint(x,zeroint()),
                        ltint(y,zeroint())
                    )
                ) = true()
                then
                    divint(x,y) = sumint(
                        succint(zeroint()),
                        divint(subint(x,y),y)
                    );
                else
                    divint(x,y) = sumint(
                        predint(zeroint()),
                        divint(sumint(x,y),y)
                    );
                endif;
                endif;
                endif;
                if gtint(m,zeroint()) = true()
                then
                    if ltint(n,zeroint()) = true()
                    then
                        modint(n,m) = modint(sumint(n,m),m);
                    else
                        modint(n,m) = subnat(n,mltnat(m,divint(n,m)) );
                    endif;
                else
                    modint(n,m) = undef;
                endif;
                eqint(x,y) = eqint(succint(x),succint(y)) );
                gtint(succint(n),n) = true();
                equivrel(eqint,int);
                irreflexive(gtint,int);
                irreflexive(ltint,int);
                transitive(gtint,int);
                transitive(ltint,int);
                transitive(geint,int);
                transitive(leint,int);
                antisymmetric(geint,int);
                antisymmetric(leint,int);
                symmetric(neint,int);
                commutative(sumint,int);
                commutative(mltint,int);
                associative(sumint,int);
                associative(mltint,int);
        end extend;
end integer;
```

```
spec character
is
    extend
        boolean
    with
        sort
            char;
        primitive
        op
            'A','B','C',...,'Z': → char;
            'a','b','c',...,'z': → char;
            '!','@','#','$','%','^','&','*','(',')': → char;
            '-','_','+','=','~','"','{','}','[',']': → char;
            '''','"',':',';',',','.','<','>','?','/': → char;
            '↑','`': → char;
            '1','2','3','4','5','6','7','8','9','0': → char;
            NUL: → char;
            SOH: → char;
            STX: → char;
            ETX: → char;
            EOT: → char;
            ENQ: → char;
            ACK: → char;
            BEL: → char;
            BS: → char;
            HT: → char;
            LF: → char;
            VT: → char;
            FF: → char;
            CR: → char;
            SO: → char;
            SI: → char;
            DLE: → char;
            DC1: → char;
            DC2: → char;
            DC3: → char;
            DC4: → char;
            NAK: → char;
            SYN: → char;
            ETB: → char;
            CAN: → char;
            EM: → char;
            SUB: → char;
            ESC: → char;
            FS: → char;
            GS: → char;
            RS: → char;
            US: → char;
            SP: → char;
            DEL: → char;
            eqchar: char,char → bool;
            gtchar: char,char → bool;
        derived
        op
            ltchar: char,char → bool;
            gechar: char,char → bool;
            lechar: char,char → bool;
            nechar: char,char → bool;
        derived
        def
            ltchar(n,m) = not(or(gtchar(n,m),eqchar(n,m)) );
```

92

```
        gechar(n,m) = not(ltchar(n,m));
        lechar(n,m) = not(gtchar(n,m));
        nechar(n,m) = not(eqchar(n,m));
  axiom
      gtchar('DEL','⁻ ') = true();
      gtchar('⁻ ','}') = true();
      gtchar('}','↑ ') = true();
      gtchar('↑ ','{') = true();
      gtchar('{','z') = true();
      gtchar('z',...,'a') = true();
      gtchar('a','"') = true();
      gtchar('"','_') = true();
      gtchar('_','⁻⁻ ') = true();
      gtchar('⁻⁻ ','↑') = true();
      gtchar('↑','′') = true();
      gtchar('′','[') = true();
      gtchar('[','Z') = true();
      gtchar('Z',...,'A') = true();
      gtchar('A','@') = true();
      gtchar('@','?') = true();
      gtchar('?','>') = true();
      gtchar('>','=') = true();
      gtchar('=','<') = true();
      gtchar('<',';') = true();
      gtchar(';',':') = true();
      gtchar(':','9') = true();
      gtchar('9',...,'0') = true();
      gtchar('0','/') = true();
      gtchar('/','.') = true();
      gtchar('.','-') = true();
      gtchar('-',',') = true();
      gtchar(',','+') = true();
      gtchar('+','*') = true();
      gtchar('*',')') = true();
      gtchar(')','(') = true();
      gtchar('(','′′') = true();
      gtchar('′′','&') = true();
      gtchar('&','%') = true();
      gtchar('%','$') = true();
      gtchar('$','#') = true();
      gtchar('#','"') = true();
      gtchar('"','!') = true();
      gtchar('!',SP) = true();
      gtchar(SP,US) = true();
      gtchar(US,RS) = true();
      gtchar(RS,GS) = true();
      gtchar(GS,FS) = true();
      gtchar(FS,ESC) = true();
      gtchar(ESC,SUB) = true();
      gtchar(SUB,EM) = true();
      gtchar(EM,CAN) = true();
      gtchar(CAN,ETB) = true();
      gtchar(ETB,SYN) = true();
      gtchar(SYN,NAK) = true();
      gtchar(NAK,DC4) = true();
      gtchar(DC4,DC3) = true();
      gtchar(DC3,DC2) = true();
      gtchar(DC2,DC1) = true();
      gtchar(DC1,DLE) = true();
      gtchar(DLE,SI) = true();
      gtchar(SI,SO) = true();
```

```
                gtchar(SO,CR) = true();
                gtchar(CR,FF) = true();
                gtchar(FF,VT) = true();
                gtchar(VT,LF) = true();
                gtchar(LF,HT) = true();
                gtchar(HT,BS) = true();
                gtchar(BS,BEL) = true();
                gtchar(BEL,ACK) = true();
                gtchar(ACK,ENQ) = true();
                gtchar(ENQ,EOT) = true();
                gtchar(EOT,ETX) = true();
                gtchar(ETX,STX) = true();
                gtchar(STX,SOH) = true();
                gtchar(SOH,NUL) = true();
                equivrel(eqchar,char);
                irreflexive(gtchar,char);
                irreflexive(ltchar,char);
                transitive(gtchar,char);
                transitive(ltchar,char);
                transitive(gechar,char);
                transitive(lechar,char);
                antisymmetric(gechar,char);
                antisymmetric(lechar,char);
                symmetric(nechar,char);
          end extend;
      end character;
```

```
spec string
parm
    extend
        boolean
    with
        sort
            lm;
        primitive
        op
            eqlm: lm,lm → bool;
            gtlm: lm,lm → bool;
        derived
        op
            ltlm: lm,lm → bool;
            gelm: lm,lm → bool;
            lelm: lm,lm → bool;
            nelm: lm,lm → bool;
        derived
        def
            ltlm(n,m) = not(or(gtlm(n,m),eqlm(n,m)) );
            gelm(n,m) = not(ltlm(n,m));
            lelm(n,m) = not(gtlm(n,m));
            nelm(n,m) = not(eqlm(n,m));
        axiom
            equivrel(eqlm,lm);
            irreflexive(gtlm,lm);
            irreflexive(ltlm,lm);
            transitive(gtlm,lm);
            transitive(ltlm,lm);
            transitive(gelm,lm);
            transitive(lelm,lm);
            antisymmetric(gelm,lm);
            antisymmetric(lelm,lm);
            symmetric(nelm,lm);
    end extend;
is
    extend
        natural,
        boolean
    with
        sort
            str;
        primitive
        op
            nullstr: → str;                          /* null string */
            makestr: lm → str;                       /* make */
            lenstr: str → nat;                       /* string length */
            headstr: str → lm;                       /* string head */
            tailstr: str → str;                      /* string tail */
            catstr: str,str → str;                   /* concatenation */
            eqstr: str,str → bool;
            gtstr: str,str → bool;
        derived
        op
            ltstr: str,str → bool;
            gestr: str,str → bool;
            lestr: str,str → bool;
            nestr: str,str → bool;
        derived
        def
            ltstr(n,m) = not(or(gtstr(n,m),eqstr(n,m)) );
```

95

```
            gestr(n,m) = not(ltstr(n,m));
            lestr(n,m) = not(gtstr(n,m));
            nestr(n,m) = not(eqstr(n,m));
        axiom
            lenstr(nullstr) = zeronat();
            lenstr(makestr(l)) = succnat(zeronat());
            lenstr(catstr(s1,s2)) = sumnat(lenstr(s1),lenstr(s2));
            headstr(makestr(l)) = l;
            tailstr(makestr(l)) = nullstr;
            headstr(catstr(makestr(l),s)) = l;
            tailstr(catstr(makestr(l),s2)) = s2;
            headstr(nullstr) = undef;
            tailstr(nullstr) = nullstr;
            catstr(catstr(s1,s2),s3) = catstr(s1,catstr(s2,s3));
            catstr(nullstr,s) = catstr(s,nullstr) = s;
            implies(eqlm(l1,l2),eqstr(makestr(l1),makestr(l2)) ) = true();
            implies(gtlm(l1,l2),gtstr(makestr(l1),makestr(l2)) ) = true();
            gtnat(lenstr(makestr(l)),lenstr(nullstr)) = true();
            implies(gtnat(lenstr(s1),lenstr(s2)),gtstr(s1,s2)) = true();
            if lenstr(s1) != zeronat()
            then
                gtnat(lenstr(catstr(s1,s2),lenstr(s2)) = true();
            else
                eqnat(lenstr(catstr(s1,s2),lenstr(s2)) = true();
            endif;
            equivrel(eqstr,str);
            irreflexive(gtstr,str);
            irreflexive(ltstr,str);
            transitive(gtstr,str);
            transitive(ltstr,str);
            transitive(gestr,str);
            transitive(lestr,str);
            antisymmetric(gestr,str);
            antisymmetric(lestr,str);
            symmetric(nestr,str);
    end extend;
end string;


spec str.chartype
is
    extend
        character
    with
        use
            string(character)
        where
            char is lm;
            eqchar is eqlm;
            gtchar is gtlm;
            ltchar is ltlm;
            gechar is gelm;
            lechar is lelm;
            nechar is nelm;
    end extend;
end str.chartype;
```

```
spec intensity
is
    extend
        boolean
    with
        sort
            intens; ·
        primitive
        op
            minintens: → intens;                          /* minimum intensity */
            maxintens: → intens;                          /* maximum intensity */
            nullintens: → intens;                         /* null intensity */
            predintens: intens → intens;
            succintens: intens → intens;
            sumintens: intens,intens → intens;
            subintens: intens,intens → intens;
            eqintens: intens,intens → bool;
            gtintens: intens,intens → bool;
        derived
    ·   op
            ltintens: intens,intens → bool;
            geintens: intens,intens → bool;
            leintens: intens,intens → bool;
            neintens: intens,intens → bool;
        derived
        def
            ltintens(n,m) = not(or(gtintens(n,m),eqintens(n,m)) );
            geintens(n,m) = not(ltintens(n,m));
            leintens(n,m) = not(gtintens(n,m));
            neintens(n,m) = not(eqintens(n,m));
        axiom
            predintens(minintens()) = undef;
            predintens(nullintens()) = undef;
            succintens(maxintens()) = undef;
            succintens(nullintens()) = undef;
            sumintens(i,nullintens()) = undef;
            subintens(i,nullintens()) = undef;
            maxintens() = [NUMINTENS]succintens(minintens())
            sumintens(i,minintens()) = i;
            subintens(i,minintens()) = i;
            sumintens(i,succintens(j)) = succintens(sumintens(i,j));
            if gtintens(i,j) = true()
            then
                subintens(i,succintens(j)) = predintens(subintens(i,j));
            else
                subintens(i,succintens(j)) = undef;
            endif;
            eqintens(i,j) = eqintens(succintens(i),succintens(j));
            eqintens(i,j) = eqintens(predintens(i),predintens(j));
            eqintens(i,succintens(predintens(i)) ) = true();
            eqintens(i,predintens(succintens(i)) ) = true();
            if or(
                eqintens(i,nullintens()),
                eqintens(j,nullintens())
            ) = true()
            then
                gtintens(i,j) = undef;
            endif;
            gtintens(succintens(i),i) = true();
            equivrel(eqintens,intens);
            irreflexive(gtintens,intens);
```

97

```
            irreflexive(ltintens,intens);
            transitive(gtintens,intens);
            transitive(ltintens,intens);
            transitive(geintens,intens);
            transitive(leintens,intens);
            antisymmetric(geintens,intens);
            antisymmetric(leintens,intens);
            symmetric(neintens,intens);
            commutative(sumintens,intens);
            associative(sumintens,intens);
      end extend;
end intensity;
```

```
spec pointcolor
is
    extend
        boolean,
        intensity
    with
        sort
            color;
        primitive
        op
            nullcolor: → color;                         /* null color */
            redcompnt: color → intens;                  /* red component */
            grncompnt: color → intens;                  /* green component */
            blucompnt: color → intens;                  /* blue component */
            eqcolor: color,color → bool;                /* equal color */
            defcolor: intens,intens,intens → color;     /* define color */
        axiom
            redcompnt(nullcolor()) = nullintens();
            grncompnt(nullcolor()) = nullintens();
            blucompnt(nullcolor()) = nullintens();
            if and(
                or(
                    or(
                        eqintens(i1,nullintens()),
                        eqintens(i2,nullintens())
                    ),
                    eqintens(i3,nullintens())
                ),
                or(
                    or(
                        not(eqintens(i1,nullintens()) ),
                        not(eqintens(i2,nullintens()) )
                    ),
                    not(eqintens(i3,nullintens()) )
                )
            ) = true()
            then
                defcolor(i1,i2,i3) = undef;
            else
                redcompnt(defcolor(i1,i2,i3)) = i1;
                grncompnt(defcolor(i1,i2,i3)) = i2;
                blucompnt(defcolor(i1,i2,i3)) = i3;
            endif;
            eqcolor(c1,c2) = and(
                and(
                    eqintens(redcompnt(c1),redcompnt(c2)),
                    eqintens(grncompnt(c1),grncompnt(c2))
                ),
                eqintens(blucompnt(c1),blucompnt(c2))
            );
            equivrel(eqcolor,color);
    end extend;
end pointcolor;
```

```
spec point
is
    extend
        boolean,
        natural,
        integer
    with
        sort
            pnt;
        primitive
        op
            xcord: pnt → int;                                    /* x coordinate */
            ycord: pnt → int;                                    /* y coordinate */
            locpnt: int,int → pnt;                               /* point location */
            eqpnt: pnt,pnt → bool;                               /* equal point */
            gtpnt: pnt,pnt → bool;                               /* right & above */
            ltpnt: pnt,pnt → bool;                               /* left & below */
            gepnt: pnt,pnt → bool;                               /* right & above, or right inline
                            or above inline */
            lepnt: pnt,pnt → bool;                               /* left & below, or left inline
                            or below inline */
            offsetpnt: int,int,pnt → pnt;                        /* point offset */
        axiom
            xcord(locpnt(i1,i2)) = i1;
            ycord(locpnt(i1,i2)) = i2;
            eqpnt(p1,p2) = and(
                eqint(xcord(p1),xcord(p2)),
                eqint(ycord(p1),ycord(p2))
            );
            gtpnt(p1,p2) = and(
                gtint(xcord(p1),xcord(p2)),
                gtint(ycord(p1),ycord(p2))
            );
            ltpnt(p1,p2) = and(
                ltint(xcord(p1),xcord(p2)),
                ltint(ycord(p1),ycord(p2))
            );
            gepnt(p1,p2) = and(
                or(
                    gtint(xcord(p1),xcord(p2)),
                    eqint(xcord(p1),xcord(p2))
                ),
                or(
                    gtint(ycord(p1),ycord(p2)),
                    eqint(ycord(p1),ycord(p2))
                )
            );
            lepnt(p1,p2) = and(
                or(
                    ltint(xcord(p1),xcord(p2)),
                    eqint(xcord(p1),xcord(p2))
                ),
                or(
                    ltint(ycord(p1),ycord(p2)),
                    eqint(ycord(p1),ycord(p2))
                )
            );
            if x = zeroint()
            then
                xcord(offsetpnt(x,y,p)) = xcord(p);
            else if gtint(x,zeroint()) = true()
```

100

```
            then
                xcord(offsetpnt(x,y,p)) = succint(xcord(offsetpnt(predint(x),y,p)) );
            else
                xcord(offsetpnt(x,y,p)) = predint(xcord(offsetpnt(succint(x),y,p)) );
            endif;
            endif;
            if y = zeroint()
            then
                ycord(offsetpnt(x,y,p)) = ycord(p2);
            else if gtint(y,zeroint()) = true()
            then
                ycord(offsetpnt(x,y,p)) = succint(ycord(offsetpnt(x,predint(y),p)) );
            else
                ycord(offsetpnt(x,y,p)) = predint(ycord(offsetpnt(x,succint(y),p)) );
            endif;
            endif;
            equivrel(eqpnt,pnt);
            reflexive(gepnt,pnt);
            reflexive(lepnt,pnt);
            irreflexive(gtpnt,pnt);
            irreflexive(ltpnt,pnt);
            transitive(gtpnt,pnt);
            transitive(ltpnt,pnt);
            transitive(gepnt,pnt);
            transitive(lepnt ,pnt);
        end extend;
end point;
```

```
spec rectangle
is
    extend
        boolean,
        integer,
        point
    with
        sort
            rct;
        primitive
        op
            origin: rct → pnt;                              /* lower left corner */
            corner: rct → pnt;                              /* upper right corner */
            xdimrct: rct → int;                             /* x dimension */
            ydimrct: rct → int;                             /* y dimension */
            area: pnt,pnt → rct;                            /* define rct */
            inrct: pnt,rct → bool;                          /* pnt inside rct test */
            disjrct: rct,rct → bool;                        /* disjoint rcts */
            intsctrct: rct,rct → rct;                       /* rct intersection */
            putrct: pnt ,rct → rct;                         /* put rct at location */
            shiftrct: int,int,rct → rct;                    /* shift rct */
        axiom
            if ltint(xcord(p2),xcord(p1)) = true()
            then
                xcord(origin(area(p1,p2)) ) = xcord(p2);
            else
            xcord(origin(area(p1,p2)) ) = xcord(p1);
            endif;
            if ltint(ycord(p2),ycord(p1)) = true()
            then
                ycord(origin(area(p1,p2)) ) = ycord(p2);
            else
                ycord(origin(area(p1,p2)) ) = ycord(p1);
            endif;
            if gtint(xcord(p1),xcord(p2)) = true()
            then
                xcord(corner(area(p1,p2)) ) = xcord(p1);
            else
                xcord(corner(area(p1,p2)) ) = xcord(p2);
            endif;
            if gtint(ycord(p1),ycord(p2)) = true()
            then
                ycord(corner(area(p1,p2)) ) = ycord(p1);
            else
                ycord(corner(area(p1,p2)) ) = ycord(p2);
            endif;
            inrct(p,r) = and(
                gepnt(p,origin(r)),
                lepnt(p,corner(r))
            );
            disjrct(r1,r2) =
                not(or(
                    or(
                        inrct(origin(r2),r1),
                        inrct(corner(r2),r1)
                    ),
                    or(
                        inrct(
                            locpnt(xcord(origin(r2)),ycord(corner(r2)) ),
                            r1
                        ),
```

102

```
                inrct(
                    locpnt(xcord(corner(r2)),ycord(origin(r2)) ),
                    r1
                )
            )
        ));
    if disjrct(r1,r2) = true()
    then
        intsctrct(r1,r2) = undef;
    else
        inrct(p,intsctrct(r1,r2) = and(
            inrct(p,r1),
            inrct(p,r2)
        );
    endif;
    shiftrct(x,y,r) = area(
        offsetpnt(x,y,origin(r)),
        offsetpnt(x,y,corner(r))
    );
    putrct(p,r) = area(
        p,
        offsetpnt(xdimrct(r),ydimrct(r),p)
    );
    xdimrct(r) = subint(
        xcord(origin(r)),
        xcord(corner(r))
    );
    ydimrct(r) = subint(
        ycord(origin(r)),
        ycord(corner(r))
    );
    end extend;
end rectangle;
```

```
spec imageform
is
    extend
        boolean,
        pointcolor,
        point,
        rectangle
    with
        sort
            form;
        primitive
        op
            initform: rct → form;                      /* initialize form */
            farea: form → rct;                         /* rct area of form */
            getcolor: pnt,form → color;                /* get pnt color */
            fillform: color,form → color;              /* fill form */
            setcolor: pnt,color,form → form;           /* set pnt color */
                /**************************************
                 * invform - inverse form
                 *    given color A, color B, form F
                 *    map F foreground colors to A
                 *    map F background to B
                 */
            invform: color,color,form → form;
        axiom
            farea(initform(r)) = r;
            getcolor(p,initform(r)) = nullcolor();
            if inrct(p,farea(f)) = true()
                then getcolor(p,setcolor(p,c,f)) = c;
                else getcolor(p,f) = nullcolor();
            endif;
            if inrct(p,farea(f)) = true() then
                getcolor(p,fillform(c,f)) = c;
            endif;
            if inrct(p,farea(f)) = false() then
                setcolor(p,c,f) = undef;
            endif;
            if inrct(p,farea(f)) = true()  then
                if getcolor(p,f) = nullcolor()
                then
                    getcolor(p,invform(c1,c2,f)) = c2;
                else
                    getcolor(p,invform(c1,c2,f)) = c1;
                endif;
            endif;
    end extend;
end imageform;
```

104

```
spec iconfont
is
    extend
        boolean,
        natural,
        pointcolor,
        rectangle,
        imageform,
        pntblktrans
    with
        sort
            font;
        primitive
        op
            initfont: rct → font;                        /* initialize font */
            rctfont: font → rct;                         /* rct of font icons */
            lenfont: font → nat;                         /* number of icons in font */
            spmap: rct,pnt → pnt;                        /* map spot (font loc) to pnt */
            psmap: rct,pnt → pnt;                        /* map pnt to spot (font loc) */
            infont: nat,font → bool;                     /* for given index, does font have icon */
            delfont: nat,font → font;                    /* delete icon from font */
            getfont: nat,font → form;                    /* get icon form from font */
            setfont: form,nat,font → font;               /* put icon into font */
            offsetfont: int,int,font,pnt → pnt;          /* offset in multiples of font rcts */
        axiom
            rctfont(initfont(r)) = r;
            lenfont(initfont(r)) = zeronat();
            spmap(r,p) = locpnt(
                mltint(xcord(p),xdimrct(r)),
                mltint(ycord(p),ydimrct(r))
            );
            psmap(r,p) = locpnt(
                divint(xcord(p),xdimrct(r)),
                divint(ycord(p),ydimrct(r))
            );
            infont(id,initfont(r)) = false();
            infont(id,delfont(id,ft)) = false();
            infont(id,setfont(f,id,ft)) = true();
            if and(
                eqint(xdimrct(rctfont(ft)),xdimrct(farea(f)) ),
                eqint(ydimrct(rctfont(ft)),ydimrct(farea(f)) )
            ) = false()
            then
                setfont(f,id,ft) = undef;
            endif;
            if infont(id,ft) = true()
            then
                lenfont(setfont(f,id,ft)) = lenfont(ft);
            else
                lenfont(setfont(f,id,ft)) = succnat(lenfont(ft));
            endif;
            if infont(id,ft) = true()
            then
                lenfont(delfont(id,ft)) = prednat(lenfont(ft));
            else
                lenfont(delfont(id,ft)) = lenfont(ft);
            endif;
            if infont(id,ft) = false() then
                getfont(id,ft) = undef;
            endif;
            rctfont(ft) = farea(getfont(id,ft));
```

105

```
            getfont(id,setfont(f,id,ft)) = f;
            getfont(id,setfont(a,id,setfont(b,id,ft)) ) = a;
            offsetfont(x,y,ft,p) = locpnt(
                sumint(xcord(p),mltint(x,xdimrct(rctfont(ft)) )),
                sumint(ycord(p),mltint(y,ydimrct(rctfont(ft)) ))
            );
        end extend;
end iconfont;
```

```
spec pntblktrans
is
    extend
        natural,
        integer,
        point,
        rectangle,
        form
    with
        sort
            ptblt;
        primitive
        op
            initptblt: → ptblt;                     /* initialize ptblt */
            getsrct: ptblt → rct;                   /* get source rct */
            getdrct: ptblt → rct;                   /* get destination rct */
            getcrct: ptblt → rct;                   /* get clipping rct */
            getrule: ptblt → nat;                   /* get copy rule */
            setsrct: rct,ptblt → ptblt;             /* set source rct */
            setdrct: rct,ptblt → ptblt;             /* set destination rct */
            setcrct: rct,ptblt → ptblt;             /* set clipping rct */
            setrule: nat,ptblt → ptblt;             /* set copy rule */
                /*****************************************
                 * copyblt - form copy operation:
                 *    given source, mask destination forms;
                 *    call cpyrecur with origin of wksrct
                 *    ptblt controls operation;
                 */
            copyblt: ptblt,form,form,form → form;
                /*****************************************
                 * drawline - draws line between two pnts:
                 *    given start pnt, stop pnt, brush, destination, mask;
                 *    calls recursive h/vdrawloop depending on slope of line
                 *    drawloop constructs line using repeated
                 *        calls to copyblt using source form as a brush
                 */
            drawline: pnt,pnt,ptblt,form,form,form → form;
                /*****************************************
                 * copyfont - copy icon from font to a given point in the dest form
                 *    the source and dest rct in ptblt are automatically set
                 */
            copyfont: pnt,ptblt,nat,font,form,form → form;
                /*****************************************
                 * invcopyfont - same as copyfont but with inverse coloring on the
                 *    the font form source.
                 */
            invcopyfont: color,color,pnt,ptblt,nat,font,form,form → form;
        hidden
        op
                /*****************************************
                 * wksrct - working source rct
                 *    intersection of source form farea
                 *    and the ptblt source rct
                 */
            wksrct: form,ptblt → rct;
                /*****************************************
                 * wkdrct - working destination rct
                 *    intersection of destination form farea
                 *    and the ptblt destination rct
                 */
            wkdrct: form,ptblt → rct;
```

107

```
/***********************************
* modpnt - pnt modulo (2D modulo):
*    given pnt P, form F
*    if P in F
*        then P
*        else (wrap P around into F)
*            reduce coord of P
*            by dim of F
*            until P in F
*/
modpnt: pnt,form → pnt;
    /***********************************
    * getmcolor - applies the masking rules
    *    given pnt P, source S, dest D, mask M, ptblt B
    *    returns color MS (masked source color)
    *    based on :
    *        masking policy
    *        S color @ P
    *        M color @ modpnt of
    *            matchpnt of P,S,D,B
    */
getmcolor: pnt,ptblt,form,form,form → color;
    /***********************************
    * nextpnt - given pnt P, returns next pnt in wksrct
    *    based on sequential ordering imposed on rct:
    *        start at origin
    *        if right neighbor of P in rct
    *            then return right neighbor of P
    *            else
    *                move left to rct boundary
    *                return pnt above
    */
nextpnt: pnt,ptblt,form → pnt;
    /***********************************
    * matchpnt - find corresponding pnt in dest
    *    given pnt P, source S, dest D, ptblt B
    *    returns pnt that is offset XY from the
    *        origin of the wkdrct
    *    where XY is the offset from the
    *        origin of the wksrct
    *        that equals P
    */
matchpnt: pnt,ptblt,form,form → pnt;
    /***********************************
    * copypnt - set color at pnt in dest
    *    given pnt P, source S, dest D, mask M, ptblt B
    *    set color @ matchpnt of P,S,D,B
    *    based on:
    *        B copy rule
    *        MS color from getmcolor of P,S,M
    *        D color @ matchpnt of P,S,D,B
    */
copypnt: pnt,ptblt,form,form,form → form;
    /***********************************
    * cpyrecur - recursive function of copyblt
    *    given pnt P
    *    if P in wksrct
    *        then
    *            call copypnt with P
    *            call cpyrecur with nextpnt of P
    *        else
```

108

```
    *          stop recursion
    */
cpyrecur: pnt,ptblt,form,form,form → form;
    /******************************************
    * hdrawloop - recursive function of drawline
    *    used when absolute value of slope is < 45 degrees
    *    walks line one horizontal point at a time
    *         moving vertically as required,
    *    at each step:
    *         sets ptblt destination rct
    *         calls copyblt
    */
hdrawloop: nat,int,int,int,int,int,form,form,form,ptblt → form;
    /******************************************
    * vdrawloop - recursive function of drawline
    *    used when absolute value of slope is >= 45 degrees
    *    walks line one vertical point at a time
    *         moving horizontally as required
    *    at each step:
    *         sets ptblt destination rct
    *         calls copyblt
    */
vdrawloop: nat,int,int,int,int,int,form,form,form,ptblt → form;
axiom
    getsrct(initptblt()) = area(
        locpnt(zeroint(),zeroint()),
        locpnt(zeroint(),zeroint())
    );
    getdrct(initptblt()) = area(
        locpnt(zeroint(),zeroint()),
        locpnt(zeroint(),zeroint())
    );
    getcrct(initptblt()) = area(
        locpnt(zeroint(),zeroint()),
        locpnt(zeroint(),zeroint())
    );
    getrule(initptblt()) = zeronat();
    getsrct(setsrct(r,pb)) = r;
    getdrct(setdrct(r,pb)) = r;
    getcrct(setcrct(r,pb)) = r;
    getrule(setrule(n,pb)) = n;
    wksrct(f,pb) = intsctrct(farea(f),getsrct(pb));
    wkdrct(f,pb) = intsctrct(farea(f),getdrct(pb));
    modpnt(p,f) = offsetpnt(
        modint(xcord(p),xdimrct(farea(f)) ),
        modint(ycord(p),ydimrct(farea(f)) ),
        origin(farea(f))
    );
        /******************************************
        * matchpnt
        *    p : pnt in source
        *    pb : ptblt
        *    s : source
        *    d : dest
        ******************************************/
    matchpnt(p,pb,s,d) = offsetpnt(
        subint(
            xcord(p),
            xcord(origin(wksrct(s,pb)) )
        ),
        subint(
```

```
                ycord(p),
                ycord(origin(wksrct(s,pb)) )
            ),
            origin(wkdrct(d,pb))
);
    /****************************************
     * getmcolor
     *    p : pnt in source
     *    pb : ptblt
     *    s : source
     *    m : mask
     *    d : destination
     ****************************************/
if or(
    eqcolor(getcolor(p,s),nullcolor()),
    eqcolor(
        getcolor(modpnt(matchpnt(p,pb,s,d),m),m),
        nullcolor()
    )
) = true()
then
    getmcolor(p,pb,s,m,d) = getcolor(p,s);
else
    getmcolor(p,pb,s,m,d) = getcolor(modpnt(matchpnt(p,pb,s,d),m),m);
endif;
    /****************************************
     * nextpnt
     *    p : pnt in source
     *    pb : ptblt
     *    s : source
     ****************************************/
if ltint(
    xcord(p),
    xcord(corner(wksrct(s,pb)) )
) = true()
then
    nextpnt(p,pb,s) = locpnt(
        succint(xcord(p)),
        ycord(p)
    );
else
    nextpnt(p,pb,s) = locpnt(
        xcord(origin(wksrct(s,pb)) ),
        succint(ycord(p))
    );
endif;
    /****************************************
     * copypnt
     *    p : pnt in source
     *    pt : ptblt
     *    s : source
     *    m : mask
     *    d : destination
     ****************************************/
if inrct(
    matchpnt(p,pb,s,d)
    intsctrct(
        wkdrct(d,pb),
        getcrct(pb)
    )
) = true()
```

110

```
    then
        if getrule(pb) = zeronat()
        then
            copypnt(p,pb,s,m,d) = d;
        else if getrule(pb) = [1]succnat(zeronat())
        then
            if and(
                not(eqcolor(getmcolor(p,pb,s,m,d),nullcolor()) ),
                not(eqcolor(
                    getcolor(
                        matchpnt(p,pb,s,d),
                        d
                    ),
                    nullcolor()
                ))
            ) = true()
            then
                copypnt(p,pb,s,m,d) = setcolor(
                    matchpnt(p,pb,s,d),
                    getmcolor(p,pb,s,m,d),
                    d
                );
            else
                copypnt(p,pb,s,m,d) = d;
            endif;
        else if getrule(pb) = [2]succnat(zeronat())
        then
            if and(
                not(eqcolor(getmcolor(p,pb,s,m,d),nullcolor()) ),
                eqcolor(
                    getcolor(
                        matchpnt(p,pb,s,d),
                        d
                    ),
                    nullcolor()
                )
            ) = true()
            then
                copypnt(p,pb,s,m,d) = setcolor(
                    matchpnt(p,pb,s,d),
                    getmcolor(p,pb,s,m,d),
                    d
                );
            else
                copypnt(p,pb,s,m,d) = d;
            endif;
        else if getrule(pb) = [3]succnat(zeronat())
        then
            if getmcolor(p,pb,s,m,d) != nullcolor()
            then
                setcolor(
                    matchpnt(p,pb,s,d),
                    getmcolor(p,pb,s,m,d),
                    d
                );
            else
                copypnt(p,pb,s,m,d) = d;
            endif;
        else if getrule(pb) = [4]succnat(zeronat())
        then
            if and(
```

111

```
                    eqcolor(getmcolor(p,pb,s,m,d),nullcolor()),
                    not(eqcolor(
                        getcolor(
                            matchpnt(p,pb,s,d),
                            d
                        ),
                        nullcolor()
                    ))
                ) = true()
                then
                    copypnt(p,pb,s,m,d) = setcolor(
                        matchpnt(p,pb,s,d),
                        nullcolor(),
                        d
                    );
                endif;
            else if getrule(pb) = [5]succnat(zeronat())
            then
                if getcolor(
                    matchpnt(p,pb,s,d),
                    d
                )) != nullcolor()
                then
                    copypnt(p,pb,s,m,d) = setcolor(
                        matchpnt(p,pb,s,d),
                        getmcolor(p,pb,s,m,d),
                        d
                    );
                else
                    copypnt(p,pb,s,m,d) = d;
                endif;
            else if getrule(pb) = [6]succnat(zeronat())
            then
                if and(
                    or(
                        eqcolor(getmcolor(p,pb,s,m,d),nullcolor()),
                        not(eqcolor(
                            getcolor(
                                matchpnt(p,pb,s,d),
                                d
                            ),
                            nullcolor()
                        ))
                    ),
                    or(
                        not(eqcolor(getmcolor(p,pb,s,m,d),nullcolor()) ),
                        eqcolor(
                            getcolor(
                                matchpnt(p,pb,s,d),
                                d
                            ),
                            nullcolor()
                        )
                    )
                ) = true()
                then
                    copypnt(p,pb,s,m,d) = setcolor(
                        matchpnt(p,pb,s,d),
                        getmcolor(p,pb,s,m,d),
                        d
                    );
```

112

```
                    else
                        copypnt(p,pb,s,m,d) = d;
                    endif;
            else if getrule(pb) = [7]succnat(zeronat())
            then
                if or(
                    not(eqcolor(getmcolor(p,pb,s,m,d),nullcolor()) ),
                    not(eqcolor(
                        getcolor(
                            matchpnt(p,pb,s,d),
                            d
                        ),
                        nullcolor()
                    ))
                ) = true()
                then
                    copypnt(p,pb,s,m,d) = setcolor(
                        matchpnt(p,pb,s,d),
                        getmcolor(p,pb,s,m,d),
                        d
                    );
                else
                    copypnt(p,pb,s,m,d) = d;
            else
                    copypnt(p,pb,s,m,d) = d;
                endif;
            endif;
            endif;
            endif;
            endif;
            endif;
            endif;
            endif;
            endif;
endif;
    /***************************************
    * cpyrecur
    *   p : pnt in source
    *   pb : ptblt
    *   s : source
    *   m : mask
    *   d : destination
    ***************************************/
if inrct(p,wksrct(s,pb)) = true()
then
    /* copy pnt and continue */
    cpyrecur(p,pb,s,m,d) = cpyrecur(
        nextpnt(p,pb,s),
        pb
        s,
        m,
        copypnt(p.pb,s,m,d),
    );
else
    /* all source pnts visited */
    cpyrecur(p,pb,s,m,d) = d;
endif;
    /***************************************
    * copyblt
    *   pb : ptblt
    *   s : source
```

```
*    m : mask
*    d : destination
****************************************/
if or(
    disjrct(farea(s),getsrct(pb)),
    disjrct(farea(d),getdrct(pb))
) = true()
then
    copyblt(pb,s,m,d) = d;
else
    copyblt(pb,s,m,d) =
        cpyrecur(
            origin(wksrct(s,pb)),
            pb
            s,
            m,
            d,
        );
endif;
    /****************************************
    * hdrawloop
    *    n : dist to go (major axis)
    *    p : minor axis move counter (vertical)
    *    dx : xDelta sign
    *    dy : yDelta sign
    *    px : yDelta abs
    *    py : xDelta abs
    *    s : source form
    *    d : dest form
    *    m : mask form
    *    pb : ptblt
    ****************************************/
/* is it the last step ? */
if n = succnat(zeronat())
then
    /* time to move in minor direction ? */
    if ltint(subint(p,px),zeroint()) = true()
    then
        /* move minor */
        hdrawloop(n,p,dx,dy,px,py,s,d,m,pb) = copyblt(
            setdrct(shiftrct(dx,dy,getdrct(pb)),pb)
            s,
            m,
            d,
        );
    else
        /* move major */
        hdrawloop(n.p,dx,dy,px,py,s,d,m,pb) = copyblt(
            setdrct(shiftrct(dx,zeroint(),getdrct(pb)),pb)
            s,
            m,
            d,
        );
    endif;
else if ltint(subint(p,px),zeroint()) = true()
then
    /* move minor and continue */
    hdrawloop(n,p,dx,dy,px,py,s,d,m,pb) = hdrawloop(
        /* reduce distance to go */
        subnat(n,succnat(zeronat())),
        /* reset counter for next minor move */
```

114

```
        sumint(subint(p,px),py),
        dx,
        dy,
        px,
        py,
        s,
        /* move minor and major then copy brush */
        copyblt(
            setdrct(shiftrct(dx,dy,getdrct(pb)),pb)
            s,
            m,
            d,
        ),
        m,
        setdrct(shiftrct(dx,dy,getdrct(pb)),pb)
    );
else
    /* move major and continue */
    hdrawloop(n,p,dx,dy,px,py,s,d,m,pb) = hdrawloop(
        /* reduce dist to go */
        subnat(n,succnat(zeronat())),
        /* reduce count till next minor move */
        subint(p,px),
        dx,
        dy,
        px,
        py,
        s,
        /* move major then copy brush */
        copyblt(
            setdrct(shiftrct(dx,zeroint(),getdrct(pb)),pb)
            s,
            m,
            d,
        ),
        m,
        setdrct(shiftrct(dx,zeroint(),getdrct(pb)),pb)
    );
endif;
endif;
    /*****************************************
    * vdrawloop
    *    n : dist to go (major axis)
    *    p : minor axis move counter (horizontal)
    *    dx : xDelta sign
    *    dy : yDelta sign
    *    px : yDelta abs
    *    py : xDelta abs
    *    s : source form
    *    d : dest form
    *    m : mask form
    *    pb : ptblt
    *****************************************/
/* is it the last step ? */
if n = succnat(zeronat())
then
    /* last step */
    if ltint(subint(p,py),zeroint()) = true()
    then
        /* move minor */
        vdrawloop(n,p,dx,dy,px,py,s,d,m,pb) = copyblt(
```

115

```
                            setdrct(shiftrct(dx,dy,getdrct(pb)),pb)
                    s,
                    m,
                    d,
                );
            else
                /* move major */
                vdrawloop(n,p,dx,dy,px,py,s,d,m,pb) = copyblt(
                    setdrct(shiftrct(zeroint(),dy,getdrct(pb)),pb)
                    s,
                    m,
                    d,
                );
            endif;
        else if ltint(subint(p,py),zeroint()) = true()
        then
            /* move minor and continue walk */
            vdrawloop(n,p,dx,dy,px,py,s,d,m,pb) = vdrawloop(
                /* reduce dist to go */
                subnat(n,succnat(zeronat()) ),
                /* set counter for next minor move */
                sumint(subint(p,py),px),
                dx,
                dy,
                px,
                py,
                s,
                /* move minor and major then copy brush */
                copyblt(
                    setdrct(shiftrct(dx,dy,getdrct(pb)),pb)
                    s,
                    m,
                    d,
                ),
                m,
                setdrct(shiftrct(dx,dy,getdrct(pb)),pb)
            );
        else
            /* move major and continue walk */
            vdrawloop(n,p,dx,dy,px,py,s,d,m,pb) = vdrawloop(
                /* reduce dist to go */
                subnat(n,succnat(zeronat()) ),
                /* reduce count till minor move */
                subint(p,py),
                dx,
                dy,
                px,
                py,
                s,
                /* move major and copy brush */
                copyblt(
                    setdrct(shiftrct(zeroint(),dy,getdrct(pb)),pb)
                    s,
                    m,
                    d,
                ),
                m,
                setdrct(shiftrct(zeroint(),dy,getdrct(pb)),pb)
            );
        endif;
    endif;
```

116

```
/*****************************************
 * drawline
 *    p1 : start pnt
 *    p2 : end pnt
 *    pb : ptblk
 *    s : brush form
 *    m : mask form
 *    d : dest form
 *****************************************/
if and(
    subint(ycord(p2),ycord(p1)),
    subint(xcord(p2),xcord(p1))
) = true()
then
/* line is a single pnt */
    drawline(p1,p2,pb,s,m,d) = copyblt(pb,s,m,d);
else if ltint(
    absint(subint(ycord(p2),ycord(p1)) ),
    absint(subint(xcord(p2),xcord(p1)) )
) = true()
then
    /* lines is horizontal */
    drawline(p1,p2,pb,s,m,d) =
        hdrawloop(
            /* distance to go */
            iton(absint(subint(xcord(p2),xcord(p1)) )),
            /* dist till minor move counter */
            divint(
                absint(subint(xcord(p2),xcord(p1)) ),
                [2]succint(zeroint())
            ),
            /* dx */
            divint(
                subint(xcord(p2),xcord(p1)),
                absint(subint(xcord(p2),xcord(p1)) )
            ),
            /* dy */
            divint(
                subint(ycord(p2),ycord(p1)),
                absint(subint(ycord(p2),ycord(p1)) )
            ),
            /* px */
            absint(subint(ycord(p2),ycord(p1)) ),
            /* py */
            absint(subint(xcord(p2),xcord(p1)) ),
            s,
            copyblt(pb,s,m,d),
            m,
            pb
        );
else
    /* line is vertical */
    drawline(p1,p2,pb,s,m,d) =
        vdrawloop(
            /* dist to go */
            iton(absint(subint(ycord(p2),ycord(p1)) )),
            /* dist till minor move counter */
            divint(
                absint(subint(ycord(p2),ycord(p1)) ),
                [2]succint(zeroint())
            ),
```

117

```
                                    /* dx */
                                    divint(
                                        subint(xcord(p2),xcord(p1)),
                                        absint(subint(xcord(p2),xcord(p1)) )
                                    ),
                                    /* dy */
                                    divint(
                                        subint(ycord(p2),ycord(p1)),
                                        absint(subint(ycord(p2),ycord(p1)) )
                                    ),
                                    /* px */
                                    absint(subint(ycord(p2),ycord(p1)) ),
                                    /* py */
                                    absint(subint(xcord(p2),xcord(p1)) ),
                                    s,
                                    copyblt(pb,s,m,d),
                                    m,
                                    pb
                                );
                        endif;
                        endif;
                        /***************************************
                         * copyfont
                         *    p : position in destination for lower left corner of source form
                         *    pb : ptblt
                         *    id : index number
                         *    ft : font with source form
                         *    m : mask
                         *    d : destination form
                         ***************************************/
                copyfont(p,pb,id,ft,m,d) = copyblt(
                    setdrct(
                        putrct(p,rctfont(ft)),
                        setsrct(rctfont(ft),pb)
                    )
                    getfont(id,ft),
                    m,
                    d,
                );
                        /***************************************
                         * invcopyfont
                         *    c1 : foreground color
                         *    c2 : background color
                         *    p : position in destination for lower left corner of source form
                         *    pb : ptblt
                         *    id : index number
                         *    ft : font with source form
                         *    m : mask
                         *    d : destination form
                         ***************************************/
                invcopyfont(c1,c2,p,pb,id,ft,m,d) = copyblt(
                    setdrct(
                        putrct(p,rctfont(ft)),
                        setsrct(rctfont(ft),pb)
                    )
                    invform(c1,c2,getfont(id,ft)),
                    m,
                    d,
                );
        end extend;
    end pntblktrans;
```

```
spec identifiers
is
    extend
        boolean
    with
        sort
            memid;
            regid;
            stkid;
            dregid;
            fid;
        primitive
        op
            idopers(memid,memseg);              /* memory seg id */
            idopers(regid,regseg);              /* register seg id */
            idopers(stkid,stkseg);              /* stack seg id */
            idopers(dregid,dregseg);            /* display register seg id */
        axiom
            idaxioms(memid,memseg);
            idaxioms(regid,regseg);
            idaxioms(stkid,stkseg);
            idaxioms(dregid,dregseg);
    end extend;
end identifiers;
```

```
spec memaddress
is
    extend
        identifiers,
        boolean
    with
        sort
            memaddr;
        primitive
        op
            startmemaddr: memid → memaddr;
            nextmemaddr: memaddr → memaddr;
            prevmemaddr: memaddr → memaddr;
            getmemid: memaddr → memid;
            offset: int,memaddr → memaddr;                /* offset from memaddr */
            eqmemaddr: memaddr,memaddr → bool;
        axiom
            prevmemaddr(startmemaddr(i)) = undef;
            prevmemaddr(nextmemaddr(m)) = m;
            nextmemaddr(prevmemaddr(m)) = m;
            offset(succint(n),m) = nextmemaddr(offset(n,m));
            if offset(n,m) = startmemaddr()
            then
                offset(predint(n),m) = undef;
            else
                offset(predint(n),m) = prevmemaddr(offset(n,m));
            endif;
            eqmemid(i,getmemid(offset(n,startmemaddr(i)) )) = true();
            eqmemaddr(startmemaddr(i1),startmemaddr(i2)) = eqmemid(i1,i2);
            eqmemaddr(startmemaddr(i),nextmemaddr(a)) = false();
            eqmemaddr(nextmemaddr(a1),nextmemaddr(a2)) = eqmemaddr(a1,a2);
            offset(zeroint(),m) = m;
            equivrel(eqmemaddr,memaddr);
    end extend;
end memaddress;
```

```
spec regaddress
is
    extend
        identifiers,
        boolean
    with
        sort
            regaddr;
        primitive
        op
            startregaddr: regid → regaddr;
            nextregaddr: regaddr → regaddr;
            prevregaddr: regaddr → regaddr;
            getregid: regaddr → regid;
            eqregaddr: regaddr,regaddr → bool;
        axiom
            prevregaddr(startregaddr(i)) = undef;
            prevregaddr(nextregaddr(m)) = m;
            nextregaddr(prevregaddr(m)) = m;
            eqregaddr(startregaddr(i1),startregaddr(i2)) = eqregid(i1,i2);
            eqregaddr(startregaddr(i),nextregaddr(a)) = false();
            eqregaddr(nextregaddr(a1),nextregaddr(a2)) = eqregaddr(a1,a2);
            equivrel(eqregaddr,regaddr);
    end extend;
end regaddress;


spec stkaddress
is
    extend
        identifiers,
        boolean
    with
        sort
            stkaddr;
        primitive
        op
            getstkid: stkaddr → stkid;
            eqstkaddr: stkaddr,stkaddr → bool;
        axiom
            eqstkaddr(nextstkaddr(a1),nextstkaddr(a2)) = eqstkaddr(a1,a2);
            equivrel(eqstkaddr,stkaddr);
    end extend;
end stkaddress;
```

121

```
spec dregaddress
is
    extend
        identifiers,
        boolean
    with
        sort
            dregaddr;
        primitive
        op
            startdregaddr: dregid → dregaddr;
            nextdregaddr: dregaddr → dregaddr;
            prevdregaddr: dregaddr → dregaddr;
            getdregid: dregaddr → dregid;
            eqdregaddr: dregaddr,dregaddr → bool;
        axiom
            prevdregaddr(startdregaddr(i)) = undef;
            prevdregaddr(nextdregaddr(m)) = m;
            nextdregaddr(prevdregaddr(m)) = m;
            eqdregaddr(startdregaddr(i1),startdregaddr(i2)) = eqdregid(i1,i2);
            eqdregaddr(startdregaddr(i),nextdregaddr(a)) = false();
            eqdregaddr(nextdregaddr(a1),nextdregaddr(a2)) = eqdregaddr(a1,a2);
            equivrel(eqdregaddr,dregaddr);
    end extend;
end dregaddress;


spec monitorattribute
is
    extend
        boolean
    with
        sort
            mattribute;
        primitive
        op
            xpixels: → mattribute;
            ypixels: → mattribute;
            hscrnsize: → mattribute;
            vscrnsize: → mattribute;
            intenscapbl: → mattribute;
            colorcapbl: → mattribute;
            backgnd: → mattribute;
            dselect: → mattribute;
            eqmattribute: mattribute,mattribute → bool;
        axiom
            equivrel(eqmattribute,mattribute);
    end extend;
end monitorattribute;
```

122

```
spec files
is
    extend
        identifiers,
        boolean
    with
        sort
            file;
        primitive
        op
            getfile: fid → file;
            eqfile: file,file → bool;
        axiom
            eqfile(getfile(i1),getfile(i2)) = eqfid(i1,i2);
            equivrel(eqfile,file);
    end extend;
end files;


spec operatorclasses
is
    sort
        mop;
        dop;
        top;
        qop;
        sop;
        oop;
        rop;
        bop;
end operatorclasses;


spec instructiontype
is
    sort
        instr;
end instructiontype;
```

```
spec typing
is
    extend
        boolean,
        natural,
        integer,
        character,
        str.chartype,
        intensity,
        pointcolor,
        point,
        rectangle,
        imageform,
        pntblktrans,
        iconfont,
        identifiers,
        memaddress,
        regaddress,
        stkaddress,
        dregaddress,
        monitorattribute,
        files,
        operatorclasses,
        instructiontype
    with
        sort
            type;
            val;
        primitive
        op
            typingopers(bool);
            typingopers(nat);
            typingopers(int);
            typingopers(char);
            typingopers(str.char);
            typingopers(intens);
            typingopers(color);
            typingopers(pnt);
            typingopers(rct);
            typingopers(form);
            typingopers(ptblt);
            typingopers(font);
            typingopers(memid);
            typingopers(regid);
            typingopers(stkid);
            typingopers(dregid);
            typingopers(fid);
            typingopers(memaddr);
            typingopers(regaddr);
            typingopers(stkaddr);
            typingopers(dregaddr);
            typingopers(mtrattr);
            typingopers(file);
            typingopers(mop);
            typingopers(dop);
            typingopers(top);
            typingopers(qop);
            typingopers(sop);
            typingopers(oop);
            typingopers(rop);
            typingopers(bop);
```

```
            typingopers(instr);
        hidden
        op
            whattype: val → type;
            eqtype: type,type → bool;
        axiom
            typingaxioms(bool);
            typingaxioms(nat);
            typingaxioms(int);
            typingaxioms(char);
            typingaxioms(str.char);
            typingaxioms(intens);
            typingaxioms(color);
            typingaxioms(pnt);
            typingaxioms(rct);
            typingaxioms(form);
            typingaxioms(ptblt);
            typingaxioms(font);
            typingaxioms(memid);
            typingaxioms(regid);
            typingaxioms(stkid);
            typingaxioms(dregid);
            typingaxioms(fid);
            typingaxioms(memaddr);
            typingaxioms(regaddr);
            typingaxioms(stkaddr);
            typingaxioms(dregaddr);
            typingaxioms(mtrattr);
            typingaxioms(file);
            typingaxioms(mop);
            typingaxioms(dop);
            typingaxioms(top);
            typingaxioms(qop);
            typingaxioms(sop);
            typingaxioms(oop);
            typingaxioms(rop);
            typingaxioms(bop);
            typingaxioms(instr);
            equivrel(eqtype,type);
        end extend;
end typing;
```

```
spec operators
is
    extend
        operatorclasses,
        typing
    with
        primitive
        op
            boolnot: → mop;
            booland: → dop;
            boolor: → dop;
            natpred: → mop;
            natsucc: → mop;
            natsum: → dop;
            natsub: → dop;
            nateq: → rop;
            natgt: → rop;
            natlt: → rop;
            intpred: → mop;
            intsucc: → mop;
            intabs: → mop;
            intntoi: → mop;
            intiton: → mop;
            intsum: → dop;
            intsub: → dop;
            intmlt: → dop;
            intdiv: → dop;
            intmod: → dop;
            inteq: → rop;
            intgt: → rop;
            intlt: → rop;
            chareq: → rop;
            chargt: → rop;
            charstrlen: → mop;
            charmakestr: → mop;
            charheadstr: → mop;
            chartailstr: → mop;
            charcatstr: → dop;
            str.chareq: → rop;
            str.chargt: → rop;
            intenspred: → mop;
            intenssucc: → mop;
            intenssum: → dop;
            intenssub: → dop;
            intenseq: → rop;
            intensgt: → rop;
            colorredcompnt: → mop;
            colorgrncompnt: → mop;
            colorblucompnt: → mop;
            colordef: → top;
            coloreq: → rop;
            pntxcord: → mop;
            pntycord: → mop;
            pntloc: → dop;
            pntoffset: → top;
            pnteq: → rop;
            pntgt: → rop;
            pntlt: → rop;
            pntge: → rop;
            pntle: → rop;
            rctorigin: → mop;
```

126

```
rctcorner: → mop;
rctxdim: → mop;
rctydim: → mop;
rctarea: → dop;
rctin: → dop; ·
rctdisj: → dop;
rctint: → dop;
rctput: → dop;
rctshift: → top;
forminit: → mop;
formfarea: → mop;
formgetcolor: → dop;
formfill: → dop;
formsetcolor: → top;
forminv: → top;
fontinit: → mop;
fontrct: → mop;
fontlen: → mop;
fontspmap: → dop;
fontpsmap: → dop;
fontin: → dop;
fontdel: → dop;
fontget: → dop;
fontset: → top;
fontoffset: → qop;
ptbltgetsrct: → mop;
ptbltgetdrct: → mop;
ptbltgetcrct: → mop;
ptbltgetrule: → mop;
ptbltsetsrct: → dop;
ptbltsetdrct: → dop;
ptbltsetcrct: → dop;
ptbltsetrule: → dop;
ptbltcopy: → qop;
ptbltdrawline: → sop;
ptbltfont: → sop;
ptbltfontinv: → oop;
isbool: → bop;
isnat: → bop;
isint: → bop;
ischar: → bop;
isstr.char: → bop;
isintens: → bop;
iscolor: → bop;
ispnt: → bop;
isrct: → bop;
isform: → bop;
isptblt: → bop;
isfont: → bop;
ismemid: → bop;
isregid: → bop;
isstkid: → bop;
isdregid: → bop;
isfid: → bop;
ismemaddr: → bop;
isregaddr: → bop;
isstkaddr: → bop;
isdregaddr: → bop;
isfile: → bop;
ismop: → bop;
isdop: → bop;
```

```
            istop: → bop;
            isqop: → bop;
            issop: → bop;
            isoop: → bop;
            isrop: → bop;
            isbop: → bop;
            isinstr: → bop;
    hidden
    op
            applymop: mop,val → val;
            applydop: dop,val,val → val;
            applytop: top,val,val,val → val;
            applyqop: qop,val,val,val,val → val;
            applysop: sop,val,val,val,val,val → val;
            applyoop: oop,val,val,val,val,val,val,val → val;
            applyrop: rop,val,val → val;
            applybop: bop,val → val;
    axiom
            applymop(boolnot(),v) = valofbool(not(atomofbool(v)) );
            applydop(booland(),v1,v2) = valofbool(and(atomofbool(v1),atomofbool(v2)) );
            applydop(boolor(),v1,v2) = valofbool(or(atomofbool(v1),atomofbool(v2)) );
            applymop(natpred(),v) = valofnat(prednat(atomofnat(v)) );
            applymop(natsucc(),v) = valofnat(succnat(atomofnat(v)) );
            applydop(natsum(),v1,v2) = valofnat(sumnat(atomofnat(v1),atomofnat(v2)) );
            applydop(natsub(),v1,v2) = valofnat(subnat(atomofnat(v1),atomofnat(v2)) );
            applymop(intpred(),v) = valofint(predint(atomofint(v);
            applymop(intsucc(),v) = valofint(succint(atomofint(v);
            applymop(intabs(),v) = valofint(absint(atomofint(v);
            applymop(intntoi(),v) = valofint(ntoi(atomofnat(v)) );
            applymop(intiton(),v) = valofnat(iton(atomofint(v)) );
            applydop(intsum(),v1,v2) = valofint(sumint(atomofint(v1),atomofint(v2)) );
            applydop(intsub(),v1,v2) = valofint(subint(atomofint(v1),atomofint(v2)) );
            applydop(intmlt(),v1,v2) = valofint(mltint(atomofint(v1),atomofint(v2)) );
            applydop(intdiv(),v1,v2) = valofint(divint(atomofint(v1),atomofint(v2)) );
            applydop(intmod(),v1,v2) = valofint(modint(atomofint(v1),atomofint(v2)) );
            applymop(charstrlen(),v) = valofnat(lenstr.char(atomofstr.char(v)) );
            applymop(charmakestr(),v) = valofstr.char(makestr.char(atomofchar(v)) );
            applymop(charheadstr(),v) = valofchar(headstr.char(atomofstr.char(v)) );
            applymop(chartailstr(),v) = valofstr.char(tailstr.char(atomofstr.char(v)) );
            applydop(charcatstr(),v1,v2) = valofstr.char(catstr.char(
                atomofstr.char(v1),
                atomofstr.char(v2)
            ));
            applymop(intenspred(),v) = valofintens(predintens(atomofintens(v);
            applymop(intenssucc(),v) = valofintens(succintens(atomofintens(v);
            applydop(intenssum(),v1,v2) = valofintens(sumintens(
                atomofintens(v1),
                atomofintens(v2)
            ));
            applydop(intenssub(),v1,v2) = valofintens(subintens(
                atomofintens(v1),
                atomofintens(v2)
            ));
            applymop(colorredcompnt(),v) = valofintens(redcompnt(atomofcolor(v)) );
            applymop(colorgrncompnt(),v) = valofintens(grncompnt(atomofcolor(v)) );
            applymop(colorblucompnt(),v) = valofintens(blucompnt(atomofcolor(v)) );
            applytop(colordef(),v1,v2.v3) = valofcolor(defcolor(
                atomofintens(v1),
                atomofintens(v2),
                atomofintens(v3)
            ));
```

128

```
applymop(pntxcord(),v) = valofint(xcord(atomofpnt(v)) );
applymop(pntycord(),v) = valofint(ycord(atomofpnt(v)) );
applydop(pntloc(),v1,v2) = valofpnt(locpnt(atomofint(v1),atomofint(v2)) );
applytop(pntoffset(),v1,v2,v3) = valofpnt(offsetpnt(
    atomofint(v1),
    atomofint(v2),
    atomofpnt(v3)
));
applymop(rctorigin(),v) = valofpnt(origin(atomofrct(v)) );
applymop(rctcorner(),v) = valofpnt(corner(atomofrct(v)) );
applymop(rctxdim(),v) = valofint(xdimrct(atomofrct(v)) );
applymop(rctydim(),v) = valofint(ydimrct(atomofrct(v)) );
applydop(rctarea(),v1,v2) = valofrct(area(atomofpnt(v1),atomofpnt(v2)) );
applydop(rctin(),v1,v2) = valofbool(inrct(atomofpnt(v1),atomofrct(v2)) );
applydop(rctdisj(),v1,v2) = valofbool(disjrct(atomofrct(v1),atomofrct(v2)) );
applydop(rctint(),v1,v2) = valofrct(intsctrct(atomofrct(v1),atomofrct(v2)) );
applydop(rctput(),v1,v2) = valofrct(putrct(atomofpnt(v1),atomofrct(v2)) );
applytop(rctshift(),v1,v2,v3) = valofrct(shiftrct(
    atomofint(v1),
    atomofint(v2),
    atomofrct(v3)
));
applymop(forminit(),v) = valofform(initform(atomofrct(v)) );
applymop(formfarea(),v) = valofrct(farea(atomofform(v)) );
applydop(formgetcolor(),v1,v2) = valofcolor(getcolor(atomofpnt(v1),atomofform(v2)) );
applydop(formfill(),v1,v2) = valofform(fillform(atomofcolor(v1),atomofform(v2)) );
applytop(formsetcolor(),v1,v2,v3) = valofform(setcolor(
    atomofpnt(v1),
    atomofcolor(v2),
    atomofform(v3)
));
applytop(forminv(),v1,v2,v3) = valofform(invform(
    atomofcolor(v1),
    atomofcolor(v2),
    atomofform(v3)
));
applymop(fontinit(),v) = valoffont(initfont(atomofrct(v)) );
applymop(fontrct(),v) = valofrct(rctfont(atomoffont(v)) );
applymop(fontlen(),v) = valofnat(lenfont(atomoffont(v)) );
applydop(fontspmap(),v1,v2) = valofpnt(spmap(atomofrct(v1),atomofpnt(v2)) );
applydop(fontpsmap(),v1,v2) = valofpnt(psmap(atomofrct(v1),atomofpnt(v2)) );
applydop(fontin(),v1,v2) = valofbool(infont(atomofnat(v1),atomoffont(v2)) );
applydop(fontdel(),v1,v2) = valoffont(delfont(atomofnat(v1),atomoffont(v2)) );
applydop(fontgetfont(),v1,v2) = valofform(getfont(atomofnat(v1),atomoffont(v2)) );
applytop(fontset(),v1,v2,v3) = valoffont(setfont(
    atomofform(v1),
    atomofnat(v2),
    atomoffont(v3)
));
applyqop(fontoffset(),v1,v2,v3,v4) = valofpnt(offsetfont(
    atomofint(v1),
    atomofint(v2),
    atomoffont(v3),
    atomofpnt(v4)
));
applymop(ptbltgetsrct(),v) = valofrct(getsrct(atomofptblt(v)) );
applymop(ptbltgetdrct(),v) = valofrct(getdrct(atomofptblt(v)) );
applymop(ptbltgetcrct(),v) = valofrct(getcrct(atomofptblt(v)) );
applymop(ptbltgetrule(),v) = valofnat(getrule(atomofptblt(v)) );
applydop(ptbltsetsrct(),v1,v2) = valofptblt(setsrct(atomofrct(v1),atomofptblt(v2)) );
applydop(ptbltsetdrct(),v1,v2) = valofptblt(setdrct(atomofrct(v1),atomofptblt(v2)) );
```

```
applydop(ptbltsetcrct(),v1,v2) = valofptblt(setcrct(atomofrct(v1),atomofptblt(v2)) );
applydop(ptbltsetrule(),v1,v2) = valofptblt(setrule(atomofnat(v1),atomofptblt(v2)) );
applyqop(ptbltcopy(),v1,v2,v3,v4) = valofform(copyblt(
    atomofptblt(v1),
    atomofform(v2),
    atomofform(v3),
    atomofform(v4)
));
applysop(ptbltdrawline(),v1,v2,v3,v4,v5,v6) = valofform(drawline(
    atomofpnt(v1),
    atomofpnt(v2),
    atomofptblt(v3),
    atomofform(v4),
    atomofform(v5),
    atomofform(v6)
));
applysop(ptbltfont(),v1,v2,v3,v4,v5,v6) = valofform(copyfont(
    atomofpnt(v1),
    atomofptblt(v2),
    atomofnat(v3),
    atomoffont(v4),
    atomofform(v5),
    atomofform(v6)
));
applyoop(ptbltfontinv(),v1,v2,v3,v4,v5,v6,v7,v8) = valofform(invcopyfont(
    atomofcolor(v1),
    atomofcolor(v2),
    atomofpnt(v3),
    atomofptblt(v4),
    atomofnat(v5),
    atomoffont(v6),
    atomofform(v7),
    atomofform(v8)
));
relop(nat,eq);
relop(nat,gt);
relop(nat,lt);
relop(int,eq);
relop(int,gt);
relop(int,lt);
relop(char,eq);
relop(char,gt);
relop(str.char,eq);
relop(str.char,gt);
relop(intens,eq);
relop(intens,gt);
relop(intens,lt);
relop(color,eq);
relop(pnt,eq);
relop(pnt,gt);
relop(pnt,lt);
relop(pnt,ge);
relop(pnt,le);
isops(bool);
isops(nat);
isops(int);
isops(char);
isops(str.char);
isops(intens);
isops(color);
isops(pnt);
```

130

```
            isops(rct);
            isops(form);
            isops(ptblt);
            isops(font);
            isops(memid);
            isops(regid);
            isops(stkid);
            isops(dregid);
            isops(fid);
            isops(memaddr);
            isops(regaddr);
            isops(stkaddr);
            isops(dregaddr);
            isops(file);
            isops(mop);
            isops(dop);
            isops(top);
            isops(qop);
            isops(sop);
            isops(oop);
            isops(rop);
            isops(bop);
            isops(instr);
      end extend;
end operators;
```

```
spec instructions
is
    extend
        natural,
        integer,
        memaddress,
        regaddress,
        stkaddress,
        dregaddress,
        operatorclasses,
        instructiontype,
        typing
    with
        primitive
        op
            org: → instr;
            extern: → instr;
            globl: → instr;
            mbegin: → instr;
            mend: → instr;
            offst: int,regaddr → instr;
            link: regaddr,nat → instr;
            unlink: regaddr → instr;
            getdwin: dregaddr,regaddr → instr;
            setdwin: regaddr,dregaddr → instr;
            getmtr: mtrattr,regaddr → instr;
            setmtr: mtrattr,regaddr → instr;
            monads: mop,regaddr → instr;
            monad: mop,regaddr,regaddr → instr;
            monadi: mop,val,regaddr → instr;
            dyads: dop,regaddr,regaddr → instr;
            dyadsi: dop,val,regaddr → instr;
            dyad: dop,regaddr,regaddr,regaddr → instr;
            dyadi: dop,val,regaddr,regaddr → instr;
            triads: top,regaddr,regaddr,regaddr → instr;
            triadsi: top,val,regaddr,regaddr → instr;
            triad: top,regaddr,regaddr,regaddr,regaddr → instr;
            triadi: top,val,regaddr,regaddr,regaddr → instr;
            quads: qop,regaddr,regaddr,regaddr,regaddr → instr;
            quad: qop,regaddr,regaddr,regaddr,regaddr,regaddr → instr;
            sexads: sop,regaddr,regaddr,regaddr,regaddr,regaddr,
                regaddr → instr;
            sexad: sop,regaddr,regaddr,regaddr,regaddr,regaddr,
                regaddr,regaddr → instr;
            octads: sop,regaddr,regaddr,regaddr,regaddr,regaddr,
                regaddr,regaddr,regaddr → instr;
            octad: sop,regaddr,regaddr,regaddr,regaddr,regaddr,
                regaddr,regaddr,regaddr,regaddr → instr;
            movi_m: val,memaddr → instr;
            movi_pcr: val,int → instr;
            movi_r: val,regaddr → instr;
            movi_ri: val,regaddr → instr;
            movi_rid: val,regaddr,int → instr;
            movi_ridn: val,regaddr,nat,int → instr;
            mov_m_m: memaddr,memaddr → instr;
            mov_m_r: memaddr,regaddr → instr;
            mov_m_ri: memaddr,regaddr → instr;
            mov_m_rid: memaddr,regaddr,int → instr;
            mov_m_ridn: memaddr,regaddr,nat,int → instr;
            mov_m_d: memaddr,dregaddr → instr;
            mov_pcr_pcr: int,int → instr;
```

132

```
mov_pcr_r: int,regaddr → instr;
mov_pcr_ri: int,regaddr→ instr;
mov_pcr_rid: int,regaddr,int → instr;
mov_pcr_ridn: int,regaddr,nat,int → instr;
mov_pcr_d: int,dregaddr → instr;
mov_r_m: regaddr,memaddr → instr;
mov_r_pcr: regaddr,int → instr;
mov_r_r: regaddr,regaddr → instr;
mov_r_ri: regaddr,regaddr → instr;
mov_r_rid: regaddr,regaddr,int → instr;
mov_r_ridn: regaddr,regaddr,nat,int → instr;
mov_r_d: regaddr,dregaddr → instr;
mov_ri_m: regaddr,memaddr → instr;
mov_ri_pcr: regaddr,int → instr;
mov_ri_r: regaddr,regaddr → instr;
mov_ri_ri: regaddr,regaddr → instr;
mov_ri_rid: regaddr,regaddr,int → instr;
mov_ri_ridn: regaddr,regaddr,nat,int → instr;
mov_ri_d: regaddr,dregaddr → instr;
mov_rid_m: regaddr,int,memaddr → instr;
mov_rid_pcr: regaddr,int,int → instr;
mov_rid_r: regaddr,int,regaddr → instr;
mov_rid_ri: regaddr,int,regaddr → instr;
mov_rid_rid: regaddr,int,regaddr,int → instr;
mov_rid_ridn: regaddr,int,regaddr,nat,int → instr;
mov_rid_d: regaddr,int,dregaddr → instr;
mov_ridn_m: regaddr,nat,int,memaddr → instr;
mov_ridn_pcr: regaddr,nat,int,int → instr;
mov_ridn_r: regaddr,nat,int,regaddr → instr;
mov_ridn_ri: regaddr,nat,int,regaddr → instr;
mov_ridn_rid: regaddr,nat,int,regaddr,int → instr;
mov_ridn_ridn: regaddr,nat,int,regaddr,int,int → instr;
mov_ridn_d: regaddr,nat,int,dregaddr → instr;
mov_d_m: dregaddr,memaddr → instr;
mov_d_pcr: dregaddr,int → instr;
mov_d_r: dregaddr,regaddr → instr;
mov_d_ri: dregaddr,regaddr → instr;
mov_d_rid: dregaddr,regaddr,int → instr;
mov_d_ridn: dregaddr,regaddr,nat,int → instr;
mov_d_d: dregaddr,dregaddr → instr;
push_i: val,stkaddr → instr;
push_m: memaddr,stkaddr → instr;
push_pcr: int,stkaddr → instr;
push_r: regaddr,stkaddr → instr;
push_ri: regaddr,stkaddr → instr;
push_rid: regaddr,int,stkaddr → instr;
push_ridn: regaddr,nat,int,stkaddr → instr;
push_d: dregaddr,stkaddr → instr;
pop_x: stkaddr → instr;
pop_m: stkaddr,memaddr → instr;
pop_pcr: stkaddr,int → instr;
pop_r: stkaddr,regaddr → instr;
pop_ri: stkaddr,regaddr → instr;
pop_rid: stkaddr,regaddr,int → instr;
pop_ridn: stkaddr,regaddr,nat,int → instr;
pop_d: stkaddr,dregaddr → instr;
nop: → instr;
stop: → instr;
jmp: memaddr → instr;
jmp_i: memaddr → instr;
jmp_r: regaddr → instr;
```

```
bra: int → instr;
bra_r: regaddr → instr;
if: relop,regaddr,regaddr,memaddr → instr;
ifi: relop,regaddr,val,memaddr → instr;
ifte: relop,regaddr,regaddr,memaddr,memaddr → instr;
iftei: relop,regaddr,val,memaddr,memaddr → instr;
if_pcr: relop,regaddr,regaddr,int → instr;
ifi_pcr: relop,regaddr,val,int → instr;
ifte_pcr: relop,regaddr,regaddr,int,int → instr;
iftei_pcr: relop,regaddr,val,int,int → instr;
test: bop,regaddr,memaddr → instr;
testm: bop,memaddr,memaddr → instr;
teste: bop,regaddr,memaddr,memaddr → instr;
testme: bop,memaddr,memaddr,memaddr → instr;
test_pcr: bop,regaddr,int → instr;
testm_pcr: bop,memaddr,int → instr;
teste_pcr: bop,regaddr,int,int → instr;
testme_pcr: bop,memaddr,int,int → instr;
jsr: memaddr,stkaddr → instr;
jsr_i: memaddr,stkaddr → instr;
jsr_r: regaddr,stkaddr → instr;
bsr: int,stkaddr → instr;
bsr_r: regaddr,stkaddr → instr;
rts: stkaddr → instr;
open: stkaddr → instr;
close: stkaddr → instr;
read: stkaddr → instr;
write: stkaddr → instr;
    end extend;
end instructions;
```

```
spec amstate
is
    extend
        boolean,
        natural,
        integer,
        str.chartype,
        memaddress,
        regaddress,
        stkaddress,
        dregaddress,
        files, .
        identifiers,
        typing
    with
        sort
            state;
        primitive
            fetchm: memaddr,state → val;                    /* memory */
            fetchr: regaddr,state → val;                    /* register */
            fetchd: dregaddr,state → val;                   /* display register */
            fetchdwin: dregaddr,state → val;                /* display window */
            fetchmtr: mtrattr,state → val;                  /* monitor attribute */
            storem: val,memaddr,state → state;
            storer: val,regaddr,state → state;
            stored: val,dregaddr,state → state;
            storedwin: val,dregaddr,state → state;
            storedmtr: val,mtrattr,state → state;
            initam: → state;                                /* initialize machine */
            initstk: stkaddr,state → state;                 /* initialize stack */
            topstk: stkaddr,state → val;                    /* get top val of stack */
            pushstk: val,stkaddr,state → state;             /* push stack */
            popstk: stkaddr,state → state;                  /* pop stack */
            lalloc: nat,state → memid;                      /* get memory block from heap */
            lfree: memid,state → state;                     /* free memory block */
            indir: nat,memaddr → memaddr;                   /* memaddr for n levels of indirection */
            infile: file,state → val;                       /* read from file */
            outfile: val,file,state → state;                /* write to file */
            openfile: str.char,file,int,int,state → state;  /* open file */
            closefile: file,state → state;                  /* close file */
            rmode: → int;                                   /* read mode */
            wmode: → int;                                   /* write mode */
            rwmode: → int;                                  /* read/write mode */
            openerr: → int;                                 /* open error */
            openok: → int;                                  /* open ok */
            valdata: → int;                                 /* file ops w/ AM sort val data */
            chardata: → int;                                /* file ops w/ character data */
        hidden
        op
                /*****************************************
                 * active - lalloc flag
                 *    true when memory block is allocated w/ lalloc
                 *    false initially and after memory block released with lfree
                 *    used to prevent offsetting into non-allocated memory
                 */
            active: memid,state → bool;
        axiom
            if whattype(v) != formtype() then
                stored(v,a,q) = undef;
            endif;
            if whattype(v) != pnttype() then
```

135

```
                    storedwin(v,a,q) = undef;
            endif;
            if whattype(v) != nattype()
            then
                    storemtr(v,xpixels(),q) = undef;
                    storemtr(v,ypixels(),q) = undef;
                    storemtr(v,hscrnsize(),q) = undef;
                    storemtr(v,vscrnsize(),q) = undef;
                    storemtr(v,intenscapbl(),q) = undef;
                    storemtr(v,colorcapbl(),q) = undef;
            endif;
            if whattype(v) != colortype() then
                    storemtr(v,backgnd(),q) = undef;
            endif;
            if whattype(v) != dregaddr() then
                    storemtr(v,dselect(),q) = undef;
            endif;
            topstk(s,initstk(s)) = undef;
            popstk(s,initstk(s)) = undef;
            popstk(s,initam()) = undef;
            stateaxioms(m,memaddr);
            stateaxioms(r,regaddr);
            stateaxioms(d,dregaddr);
            stateaxioms(dwin,dregaddr);
            stateaxioms(mtr,mtrattr);
            topstk(s,pushstk(v,s,q)) = v;
            popstk(s,pushstk(v,s,q)) = q;
            active(m,initam()) = false;
            active(lalloc(n,q),q) = true;
            active(m,lfree(m,q)) = false;
            active(m,storer(v,a,q)) = active(m,q);
            active(m,storem(v,a,q)) = active(m,q);
            active(m,stored(v,a,q)) = active(m,q);
            active(m,storedwin(v,a,q)) = active(m,q);
            active(m,storexscrnsize(v,a,q)) = active(m,q);
            active(m,storeyscrnsize(v,a,q)) = active(m,q);
            active(m,storeintenscapbl(v,a,q)) = active(m,q);
            active(m,storecolorcapbl(v,a,q)) = active(m,q);
            active(m,storebackgnd(v,a,q)) = active(m,q);
            active(m,storedregaddr(v,a,q)) = active(m,q);
            active(m,initstk(a,q)) = active(m,q);
            active(m,pushstk(v,a,q)) = active(m,q);
            active(m,popstk(a,q)) = active(m,q);
            active(m,outfile(v,f,q)) = active(m,q);
            active(m,openfile(s,f,x,y,q)) = active(m,q);
            active(m,closefile(f,q)) = active(m,q);
            if active(m,q) = false() then
                    fetchm(offset(n,m),q) = undef;
            endif;
            if active(m.q) = false() then
                    storem(offset(n,m),q) = undef;
            endif;
            if ltint(n,ntoi(n2)) = true()
            then
                    offset(n,offset(n1,startmemaddr(lalloc(n2,q)) )) =
                            offset(
                                    sumint(n,n1),
                                    startmemaddr(lalloc(n2,q))
                            );
            else
                    offset(n,offset(n1,startmemaddr(lalloc(n2,q)) )) =
```

```
                    undef;
            indir(zeronat(),m) = m;
            if whattype(fetchm(indir(n,m),q)) = typememaddr()
            then
                indir(succnat(n),m) = atomofmemaddr(fetchm(indir(n,m),q));
            else
                indir(succnat(n),m) = undef;
            endif;
            openfile(s,f,n,openfile(s,f,m,x,q)) = undef;
            closefile(f,openfile(s,f,n,x,q)) = q;
            infile(f,initam()) = undef;
            infile(f,close(d,q)) = undef;
            infile(f,openfile(s,f,wmode(),x,q)) = undef;
            outfile(v,f,initam()) = undef;
            outfile(v,f,close(f,q)) = undef;
            outfile(v,f,openfile(s,f,rmode(),x,q)) = undef;
            outfile(f,openfile(s,f,m,chardata(),q)) = undef;
        end extend;
end amstate;


spec displaywindow
is
    extend
        rectangle,
        dregaddress
    with
        primitive
        op
            dwin: dregaddr → rct;
        axiom
            xdimrct(dwin(a)) = [DISPLAYSIZE]succint(zeroint());
            ydimrct(dwin(a)) = [DISPLAYSIZE]succint(zeroint());
            origin(dwin(a)) = atomofpnt(fetchdwin(a,q));
    end extend;
end displaywindow;
```

```
spec am
is
    extend
        memaddress,
        instructiontype,
        typing,
        amstate
    with
        primitive
        op
            /****************************************
             * prog - AM execution
             *   corecursive - calls xeq
             */
            prog: memaddr,state → state;
        hidden
        op
            /****************************************
             * cond - implements conditionals
             *   returns one of two input memaddrs
             *       based on bool value
             */
            cond: val,memaddr,memaddr → memaddr;
            /****************************************
             * xeq - corecursive function
             *   calls prog
             *   used for AM execution
             */
            xeq: instr,memaddr,state → state;
        axiom
            prog(a,q) = xeq(atomofinstr(fetchm(a,q)),a,q);
            cond(valofbool(true()),a1,a2) = a1;
            cond(valofbool(false()),a1,a2) = a2;
            xeq(offst(i,r),m,q) =
                prog(
                    nextmemaddr(m),
                    storer(
                        valofmemaddr(offset(i,atomofmemaddr(fetchr(r,q)) )),
                        r,
                        q
                    )
                );
            xeq(link(r,n),m,q) =
                prog(
                    nextmemaddr(m),
                    storer(
                        valofmemaddr(startmemaddr(lalloc(n,q)) ),
                        r,
                        storem(
                            fetchr(r,q),
                            startmemaddr(lalloc(n,q),q)
                        )
                    )
                );
            xeq(unlink(r),m,q) =
                prog(
                    nextmemaddr(m),
                    lfree(
                        getmemid(atomofmemaddr(fetchr(r,q)) ),
                        storer(
                            fetchm(atomofmemaddr(fetchr(r,q)),q),
```

138

```
                    r,
                    q
                )
            )
        );
    xeq(getdwin(d,r),m,q) =
        prog(
            nextmemaddr(m),
            storer(
                fetchdwin(d,q),
                r,
                q
            )
        );
    xeq(setdwin(r,d),m,q) =
        prog(
            nextmemaddr(m),
            storedwin(
                fetchr(r,q),
                d,
                q
            )
        );
    xeq(getmtr(t,r),m,q) =
        prog(
            nextmemaddr(m),
            storer(
                fetchmtr(t,q),
                r,
                q
            )
        );
    xeq(setmtr(r,t),m,q) =
        prog(
            nextmemaddr(m),
            storemtr(
                fetchr(r,q),
                t,
                q
            )
        );
    xeq(monads(o,r1),m,q) =
        prog(
            nextmemaddr(m),
            storer(
                applymop(
                    o,
                    fetchr(r1,q)
                ),
                r1,
                q
            )
        );
    xeq(monad(o,r1,r2),m,q) =
        prog(
            nextmemaddr(m),
            storer(
                applymop(
                    o,
                    fetchr(r1,q)
                ),
```

```
                r2,
                q
            )
        );
    xeq(monadi(o,v,r1),m,q) =
        prog(
            nextmemaddr(m),
            storer(
                applymop(o,v),
                r1,
                q
            )
        );
    xeq(dyads(o,r1,r2),m,q) =
        prog(
            nextmemaddr(m),
            storer(
                applydop(
                    o,
                    fetchr(r1,q),
                    fetchr(r2,q)
                ),
                r2,
                q
            )
        );
    xeq(dyadsi(o,v,r1),m,q) =
        prog(
            nextmemaddr(m),
            storer(
                applydop(
                    o,
                    v,
                    fetchr(r1,q)
                ),
                r1,
                q
            )
        );
    xeq(dyad(o,r1,r2,r3),m,q) =
        prog(
            nextmemaddr(m),
            storer(
                applydop(
                    o,
                    fetchr(r1,q),
                    fetchr(r2,q)
                ),
                r3,
                q
            )
        );
    xeq(dyadi(o,v,r1,r2),m,q) =
        prog(
            nextmemaddr(m),
            storer(
                applydop(
                    o,
                    v,
                    fetchr(r1,q)
                ),
```

```
                r2,
                q
            )
        );
xeq(triads(o,r1,r2,r3),m,q) =
    prog(
        nextmemaddr(m),
        storer(
            applytop(
                o,
                fetchr(r1,q),
                fetchr(r2,q),
                fetchr(r3,q)
            ),
            r3,
            q
        )
    );
xeq(triadsi(o,v,r1,r2),m,q) =
    prog(
        nextmemaddr(m),
        storer(
            applytop(
                o,
                v,
                fetchr(r1,q),
                fetchr(r2,q)
            ),
            r2,
            q
        )
    );
xeq(triad(o,r1,r2,r3,r4),m,q) =
    prog(
        nextmemaddr(m),
        storer(
            applytop(
                o,
                fetchr(r1,q),
                fetchr(r2,q),
                fetchr(r3,q)
            ),
            r4,
            q
        )
    );
xeq(triadi(o,v,r1,r2,r3),m,q) =
    prog(
        nextmemaddr(m),
        storer(
            applytop(
                o,
                v,
                fetchr(r1,q),
                fetchr(r2,q)
            ),
            r3,
            q
        )
    );
xeq(quads(o,r1,r2,r3,r4),m,q) =
```

```
            prog(
                nextmemaddr(m),
                storer(
                    applyqop(
                        o,
                        fetchr(r1,q),
                        fetchr(r2,q),
                        fetchr(r3,q),
                        fetchr(r4,q)
                    ),
                    r4,
                    q
                )
            );
xeq(quad(o,r1,r2,r3,r4,r5),m,q) =
            prog(
                nextmemaddr(m),
                storer(
                    applyqop(
                        o,
                        fetchr(r1,q),
                        fetchr(r2,q),
                        fetchr(r3,q),
                        fetchr(r4,q)
                    ),
                    r5,
                    q
                )
            );
xeq(sexads(o,r1,r2,r3,r4,r5,r6),m,q) =
            prog(
                nextmemaddr(m),
                storer(
                    applysop(
                        o,
                        fetchr(r1,q),
                        fetchr(r2,q),
                        fetchr(r3,q),
                        fetchr(r4,q),
                        fetchr(r5,q),
                        fetchr(r6,q)
                    ),
                    r6,
                    q
                )
            );
xeq(sexad(o,r1,r2,r3,r4,r5,r6,r7),m,q) =
            prog(
                nextmemaddr(m),
                storer(
                    applysop(
                        o,
                        fetchr(r1,q),
                        fetchr(r2,q),
                        fetchr(r3,q),
                        fetchr(r4,q),
                        fetchr(r5,q),
                        fetchr(r6,q)
                    ),
                    r7,
                    q
```

```
            )
    );
xeq(octads(o,r1,r2,r3,r4,r5,r6,r7,r8),m,q) =
    prog(
        nextmemaddr(m),
        storer(
            applyoop(
                o,
                fetchr(r1,q),
                fetchr(r2,q),
                fetchr(r3,q),
                fetchr(r4,q),
                fetchr(r5,q),
                fetchr(r6,q),
                fetchr(r7,q),
                fetchr(r8,q)
            ),
            r8,
            q
        )
    );
xeq(octad(o,r1,r2,r3,r4,r5,r6,r7,r8,r9),m,q) =
    prog(
        nextmemaddr(m),
        storer(
            applyoop(
                o,
                fetchr(r1,q),
                fetchr(r2.q),
                fetchr(r3,q),
                fetchr(r4,q),
                fetchr(r5,q),
                fetchr(r6,q),
                fetchr(r7,q),
                fetchr(r8,q)
            ),
            r9,
            q
        )
    );
xeq(movi_m(v,m1),m,q) =
    prog(
        nextmemaddr(m),
        storem(v,m1,q)
    );
xeq(movi_pcr(v,i),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            v,
            offset(i,m),
            q
        )
    );
xeq(movi_r(v,r),m,q) =
    prog(nextmemaddr(m),storer(v,r,q));
xeq(movi_ri(v,r),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            v,
```

```
                    atomofmemaddr(fetchr(r,q)),
                    q
                )
            );
    xeq(movi_rid(v,r,n),m,q) =
        prog(
            nextmemaddr(m),
            storem(
                v,
                offset(
                    n,
                    atomofmemaddr(fetchr(r,q))
                ),
                q
            )
        );
    xeq(movi_ridn(v,r,i1,i2),m,q) =
        prog(
            nextmemaddr(m),
            storem(
                v,
                offset(
                    i2,
                    indir(
                        i1,
                        atomofmemaddr(fetchr(r,q))
                    )
                ),
                q
            )
        );
    xeq(movi_d(v,r),m,q) =
        prog(nextmemaddr(m),stored(v,r,q));
    xeq(mov_m_m(m1,m2),m,q) =
        prog(
            nextmemaddr(m),
            storem(
                fetchm(m1,q),
                m2,
                q
            )
        );
    xeq(mov_m_r(m1,r),m,q) =
        prog(nextmemaddr(m),storer(fetchm(m1,q),r,q));
    xeq(mov_m_ri(m1,r),m,q) =
        prog(
            nextmemaddr(m),
            storem(
                fetchm(m1,q),
                atomofmemaddr(fetchr(r,q))
                ,q
            )
        );
    xeq(mov_m_rid(m1,r,n),m,q) =
        prog(
            nextmemaddr(m),
            storem(
                fetchm(m1,q),
                offset(
                    n,
                    atomofmemaddr(fetchr(r,q))
```

144

```
                ),
                q
            )
        );
    xeq(mov_m_ridn(m1,r,i1,i2),m,q) =
        prog(
            nextmemaddr(m),
            storem(
                fetchm(m1,q),
                offset(
                    i2,
                    indir(
                        i1,
                        atomofmemaddr(fetchr(r,q))
                    )
                ),
                q
            )
        );
    xeq(mov_pcr_pcr(i1,i2),m,q) =
        prog(
            nextmemaddr(m),
            storem(
                fetchm(offset(i1,m),q),
                offset(i2,m),
                q
            )
        );
    xeq(mov_pcr_r(i,r),m,q) =
        prog(
            nextmemaddr(m),
            storer(
                fetchm(offset(i,m),q),
                r,
                q
            )
        );
    xeq(mov_pcr_ri(i,r),m,q) =
        prog(
            nextmemaddr(m),
            storem(
                fetchm(offset(i,m),q),
                atomofmemaddr(fetchr(r,q)),
                q
            )
        );
    xeq(mov_pcr_rid(i1,r,i2),m,q) =
        prog(
            nextmemaddr(m),
            storem(
                fetchm(offset(i1,m),q),
                offset(
                    i2,
                    atomofmemaddr(fetchr(r,q))
                ),
                q
            )
        );
    xeq(mov_pcr_ridn(i1,r,n,i2),m,q) =
        prog(
            nextmemaddr(m),
```

```
                        storem(
                            fetchm(offset(i1,m),q),
                            offset(
                                i2,
                                indir(
                                    n,
                                    atomofmemaddr(fetchr(r,q))
                                )
                            ),
                            q
                        )
                    );
xeq(mov_m_d(m1,r),m,q) =
    prog(nextmemaddr(m),stored(fetchm(m1,q),r,q));
xeq(mov_r_m(r,m1),m,q) =
    prog(nextmemaddr(m),storem(fetchr(r,q),m1,q));
xeq(mov_r_pcr(r,i),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchr(r,q),
            offset(i,m),
            q
        )
    );
xeq(mov_r_r(r1,r2),m,q) =
    prog(nextmemaddr(m),storer(fetchr(r1,q),r2,q));
xeq(mov_r_ri(r1,r2),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchr(r1,q),
            atomofmemaddr(fetchr(r2,q)),
            q
        )
    );
xeq(mov_r_rid(r1,r2,n),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchr(r1,q),
            offset(
                n,
                atomofmemaddr(fetchr(r2,q))
            ),
            q
        )
    );
xeq(mov_r_ridn(r1,r2,i1,i2),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchr(r1,q),
            offset(
                i2,
                indir(
                    i1,
                    atomofmemaddr(fetchr(r2,q))
                )
            ),
            q
```

```
            )
    );
xeq(mov_r_d(r1,r2),m,q) =
    prog(nextmemaddr(m),stored(fetchr(r1,q),r2,q));
xeq(mov_ri_m(r,m1),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchm(atomofmemaddr(fetchr(r,q)),q),
            m1,
            q
        )
    );
xeq(mov_ri_pcr(r,i),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchm(atomofmemaddr(fetchr(r,q)),q),
            offset(i,m),
            q
        )
    );
xeq(mov_ri_r(r1,r2),m,q) =
    prog(
        nextmemaddr(m),
        storer(
            fetchm(atomofmemaddr(fetchr(r1,q)) ),
            r2,
            q
        )
    );
xeq(mov_ri_ri(r1,r2),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchm(atomofmemaddr(fetchr(r1,q)),q),
            atomofmemaddr(
                fetchr(r2,q)
            ),
            q
        )
    );
xeq(mov_ri_rid(r1,r2,n),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchm(atomofmemaddr(fetchr(r1,q)),q),
            offset(
                n,
                atomofmemaddr(fetchr(r2,q))
            ),
            q
        )
    );
xeq(mov_ri_ridn(r1,r2,i1,i2),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchm(atomofmemaddr(fetchr(r1,q)),q),
            offset(
                i2,
```

147

```
                indir(
                    i1,
                    atomofmemaddr(fetchr(r2,q))
                )
            ),
            q
        )
    );
xeq(mov_ri_d(r1,r2),m,q) =
    prog(
        nextmemaddr(m),
        stored(
            fetchm(atomofmemaddr(fetchr(r1,q)) ),
            r2,
            q
        )
    );
xeq(mov_rid_m(r,i,m1),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchm(
                offset(
                    i,
                    atomofmemaddr(fetchr(r,q))
                ),
                q
            ),
            m1,
            q
        )
    );
xeq(mov_rid_pcr(r,i1,i2),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchm(
                offset(
                    i1,
                    atomofmemaddr(fetchr(r,q))
                ),
                q
            ),
            offset(i2,m),
            q
        )
    );
xeq(mov_rid_r(r1,n,r2),m,q) =
    prog(
        nextmemaddr(m),
        storer(
            fetchm(
                offset(
                    n,
                    atomofmemaddr(fetchr(r1,q))
                ),
                q
            ),
            r2,
            q
        )
```

```
        );
    xeq(mov_rid_ri(r1,i,r2),m,q) =
        prog(
            nextmemaddr(m),
            storem(
                fetchm(
                    offset(
                        i,
                        atomofmemaddr(fetchr(r1,q))
                    ),
                    q
                ),
                atomofmemaddr(fetchr(r2,q)),
                q
            )
        );
    xeq(mov_rid_rid(r1,i1,r2,i2),m,q) =
        prog(
            nextmemaddr(m),
            storem(
                fetchm(
                    offset(
                        i1,
                        atomofmemaddr(fetchr(r1,q))
                    ),
                    q
                ),
                offset(
                    i2,
                    atomofmemaddr(fetchr(r2,q))
                ),
                q
            )
        );
    xeq(mov_rid_ridn(r1,i1,r2,i2,i3),m,q) =
        prog(
            nextmemaddr(m),
            storem(
                fetchm(
                    offset(
                        i1,
                        atomofmemaddr(fetchr(r1,q))
                    ),
                    q
                ),
                offset(
                    i3,
                    indir(
                        i2,
                        atomofmemaddr(fetchr(r2,q))
                    )
                ),
                q
            )
        );
    xeq(mov_rid_d(r1,n,r2),m,q) =
        prog(
            nextmemaddr(m),
            stored(
                fetchm(
                    offset(
```

```
                                    n,
                                    atomofmemaddr(fetchr(r1,q))
                        ),
                        q
                    ),
                    r2,
                    q
                )
            );
    xeq(mov_ridn_m(r,n,i,m1),m,q) =
        prog(
            nextmemaddr(m),
            storem(
                fetchm(
                    offset(
                        i,
                        indir(
                            n,atomofmemaddr(fetchr(r,q))
                        )
                    ),
                    q
                ),
                q
            )
        );
    xeq(mov_ridn_pcr(r,n,i1,i2),m,q) =
        prog(
            nextmemaddr(m),
            storem(
                fetchm(
                    offset(
                        i1,
                        indir(
                            n,
                            atomofmemaddr(fetchr(r,q))
                        )
                    ),
                    q
                ),
                offset(i2,m),
                q
            )
        );
    xeq(mov_ridn_r(r1,i1,i2,r2),m,q) =
        prog(
            nextmemaddr(m),
            storer(
                fetchm(
                    offset(
                        i2,
                        indir(
                            i1,
                            atomofmemaddr(fetchr(r1,q))
                        )
                    ),
                    q
                ),
                r2,
                q
            )
        );
```

150

```
xeq(mov_ridn_ri(r1,i1,i2,r2),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchm(
                offset(
                    i2,
                    indir(
                        i1,
                        atomofmemaddr(fetchr(r1,q))
                    )
                ),
                q
            ),
            atomofmemaddr(
                fetchr(r2,q)
            ),
            q
        )
    );
xeq(mov_ridn_rid(r1,i1,i2,r2,i3),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchm(
                offset(
                    i2,
                    indir(
                        i1,
                        atomofmemaddr(fetchr(r1,q))
                    )
                ),
                q
            ),
            offset(
                i3,
                atomofmemaddr(fetchr(r2,q))
            ),
            q
        )
    );
xeq(mov_ridn_ridn(r1,i1,i2,r2,i3,i4),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchm(
                offset(
                    i2,
                    indir(
                        i1,
                        atomofmemaddr(fetchr(r1,q))
                    )
                ),
                q
            ),
            offset(
                i4,
                indir(
                    i3,
                    atomofmemaddr(fetchr(r2,q))
                )
            )
```

151

```
            ),
            q
        )
    );
xeq(mov_ridn_d(r1,i1,i2,r2),m,q) =
    prog(
        nextmemaddr(m),
        stored(
            fetchm(
                offset(
                    i2,
                    indir(
                        i1,
                        atomofmemaddr(fetchr(r1,q))
                    )
                ),
                q
            ),
            r2,
            q
        )
    );
xeq(mov_d_m(r,m1),m,q) =
    prog(nextmemaddr(m),storem(fetchd(r,q),m1,q));
xeq(mov_d_pcr(r,i),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchd(r,q),
            offset(i,m),
            q
        )
    );
xeq(mov_d_r(r1,r2),m,q) =
    prog(nextmemaddr(m),storer(fetchd(r1,q),r2,q));
xeq(mov_d_ri(r1,r2),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchd(r1,q),
            atomofmemaddr(fetchr(r2,q)),
            q
        )
    );
xeq(mov_d_rid(r1,r2,n),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchd(r1,q),
            offset(
                n,
                atomofmemaddr(fetchr(r2,q))
            ),
            q
        )
    );
xeq(mov_d_ridn(r1,r2,i1,i2),m,q) =
    prog(
        nextmemaddr(m),
        storem(
            fetchd(r1,q),
```

```
                    offset(
                        i2,
                        indir(
                            i1,
                            atomofmemaddr(fetchr(r2,q))
                        )
                    ),
                    q
                )
        );
xeq(mov_d_d(r1,r2),m,q) =
    prog(nextmemaddr(m),stored(fetchd(r1,q),r2,q));
xeq(push_i(v,s),m,q) =
    prog(nextmemaddr(m),pushstk(v,s,q));
xeq(push_m(m1,s),m,q) =
    prog(nextmemaddr(m),pushstk(fetchm(m1,q),s,q));
xeq(push_pcr(i,s),m,q) =
    prog(
        nextmemaddr(m),
        pushstk(
            fetchm(offset(i,m),q),
            s,
            q
        )
    );
xeq(push_r(r,s),m,q) =
    prog(nextmemaddr(m),pushstk(fetchr(r,q),s,q));
xeq(push_ri(r,s),m,q) =
    prog(
        nextmemaddr(m),
        pushstk(
            fetchm(atomofmemaddr(fetchr(r,q)),q),
            s,
            q
        )
    );
xeq(push_rid(r,n,s),m,q) =
    prog(
        nextmemaddr(m),
        pushstk(
            fetchm(
                offset(
                    n,
                    atomofmemaddr(fetchr(r,q))
                ),
                q
            ),
            s,
            q
        )
    );
xeq(push_ridn(r,i1,i2,s),m,q) =
    prog(
        nextmemaddr(m),
        pushstk(
            fetchm(
                offset(
                    i2,
                    indir(
                        i1,
                        atomofmemaddr(fetchr(r,q))
```

```
                )
            ),
            q
        ),
        s,
        q
    )
);
xeq(push_d(r,s),m,q) =
    prog(nextmemaddr(m),pushstk(fetchd(r,q),s,q));
xeq(pop_x(s),m,q) =
    prog(nextmemaddr(m),popstk(s,q));
xeq(pop_m(s,m1),m,q) =
    prog(
        nextmemaddr(m),
        popstk(
            s,
            storem(
                topstk(s,q),
                m1,
                q
            )
        )
    );
xeq(pop_pcr(s,i),m,q) =
    prog(
        nextmemaddr(m),
        popstk(
            s,
            storem(
                topstk(s,q),
                offset(i,m),
                q
            )
        )
    );
xeq(pop_r(s,r),m,q) =
    prog(
        nextmemaddr(m),
        popstk(
            s,
            storer(
                topstk(s,q),
                r,
                q
            )
        )
    );
xeq(pop_ri(s,r),m,q) =
    prog(
        nextmemaddr(m),
        popstk(
            s,
            storem(
                topstk(s,q),
                atomofmemaddr(fetchr(r,q)),
                q
            )
        )
    );
xeq(pop_rid(s,r,n),m,q) =
```

```
        prog(
            nextmemaddr(m),
            popstk(
                s,
                storem(
                    topstk(s,q),
                    offset(
                        n,
                        atomofmemaddr(fetchr(r,q))
                    ),
                    q
                )
            )
        );
xeq(pop_ridn(s,r,i1,i2),m,q) =
    prog(
        nextmemaddr(m),
        popstk(
            s,
            storem(
                topstk(s,q),
                offset(
                    i2,
                    indir(
                        i1,
                        atomofmemaddr(fetchr(r,q))
                    )
                ),
                q
            )
        )
    );
xeq(pop_d(s,r),m,q) =
    prog(
        nextmemaddr(m),
        popstk(
            s,
            stored(
                topstk(s,q),
                r,
                q
            )
        )
    );
xeq(nop,m,q) = prog(nextmemaddr(m),q);
xeq(stop,m,q) = prog(m,q) = q;
xeq(jmp(m1),m,q) = prog(m1,q);
xeq(jmp_i(m1),m,q) = prog(atomofmemaddr(fetchm(m1,q)),q);
xeq(jmp_r(r),m,q) = prog(atomofmemaddr(fetchr(r,q)),q);
xeq(bra(n),m,q) = prog(offset(n,nextmemaddr(m)),q);
xeq(bra_r r,m,q) = prog(offset(atomofint(fetchr(r,q)),nextmemaddr(m)),q);
xeq(if(o,r1,r2,m1),m.q) =
    prog(
        cond(
            applyrop(
                o,
                fetchr(r1,q),
                fetchr(r2,q)
            ),
            m1,
            nextmemaddr(m)
```

```
        ),
        q
    );
xeq(ifi(o,r,v,m1),m,q) =
    prog(
        cond(
            applyrop(
                o,
                fetchr(r,q),
                v
            ),
            m1,
            nextmemaddr(m)
        ),
        q
    );
xeq(ifte(o,r1,r2,m1,m2),m,q) =
    prog(
        cond(
            applyrop(
                o,
                fetchr(r1,q),
                fetchr(r2,q)
            ),
            m1,
            m2
        ),
        q
    );
xeq(iftei(o,r,v,m1,m2),m,q) =
    prog(
        cond(
            applyrop(
                o,
                fetchr(r,q),
                v
            ),
            m1,
            m2
        ),
        q
    );
xeq(if_pcr(o,r1,r2,n),m,q) =
    prog(
        cond(
            applyrop(
                o,
                fetchr(r1,q),
                fetchr(r2,q)
            ),
            offset(n,nextmemaddr(m)),
            nextmemaddr(m)
        ),
        q
    );
xeq(ifi_pcr(o,r,v,n),m,q) =
    prog(
        cond(
            applyrop(
                o,
                fetchr(r,q),
```

```
                v
            ),
            offset(n,nextmemaddr(m)),
            nextmemaddr(m)
        ),
        q
    );
xeq(ifte_pcr(o,r1,r2,i1,i2),m,q) =
    prog(
        cond(
            applyrop(
                o,
                fetchr(r1,q),
                fetchr(r2,q)
            ),
            offset(i1,nextmemaddr(m)),
            offset(i2,nextmemaddr(m))
        ),
        q
    );
xeq(iftei_pcr(o,r,v,i1,i2),m,q) =
    prog(
        cond(
            applyrop(
                o,
                fetchr(r,q),
                v
            ),
            offset(i1,nextmemaddr(m)),
            offset(i2,nextmemaddr(m))
        ),
        q
    );
xeq(test(o,r1,m1),m,q) =
    prog(
        cond(
            applybop(o,fetchr(r1,q)),
            m1,
            nextmemaddr(m)
        ),
        q
    );
xeq(testm(o,m2,m1),m,q) =
    prog(
        cond(
            applybop(o,fetchm(m2,q)),
            m1,
            nextmemaddr(m)
        ),
        q
    );
xeq(teste(o,r1,m1,m2),m,q) =
    prog(cond(applybop(o,fetchr(r1,q)),m1,m2),q);
xeq(testme(o,m3,m1,m2),m,q) =
    prog(cond(applybop(o,fetchm(m3,q)),m1,m2),q);
xeq(test_pcr(o,r1,n),m,q) =
    prog(
        cond(
            applybop(o,fetchr(r1,q)),
            offset(n,nextmemaddr(m)),
            nextmemaddr(m);
```

```
        ),
        q
    );
xeq(testm_pcr(o,m2,n),m,q) =
    prog(
        cond(
            applybop(o,fetchm(m2,q)),
            offset(n,nextmemaddr(m)),
            nextmemaddr(m)
        ),
        q
    );
xeq(teste_pcr(o,r1,i1,i2),m,q) =
    prog(
        cond(
            applybop(o,fetchr(r1,q)),
            offset(i1,nextmemaddr(m)),
            offset(i2,nextmemaddr(m))
        ),
        q
    );
xeq(testme_pcr(o,m3,i1,i2),m,q) =
    prog(
        cond(
            applybop(o,fetchm(m3,q)),
            offset(i1,nextmemaddr(m)),
            offset(i2,nextmemaddr(m))
        ),
        q
    );
xeq(jsr(m1,s),m,q) =
    prog(m1,pushstk(valofmemaddr(nextmemaddr(m)),s,q));
xeq(jsr_i(m1,s),m,q) =
    prog(
        atomofmemaddr(fetchm(m1,q)),
        pushstk(valofmemaddr(nextmemaddr(m)),s,q)
    );
xeq(jsr_r(r,s),m,q) =
    prog(
        atomofmemaddr(fetchr(r,q)),
        pushstk(valofmemaddr(nextmemaddr(m)),s,q)
    );
xeq(bsr(n,s),m,q) =
    prog(
        offset(n,nextmemaddr(m)),
        pushstk(valofmemaddr(nextmemaddr(m)),s,q)
    );
xeq(bsr_r(r,s),m,q) =
    prog(
        offset(
            atomofint(fetchr(r,q)),
            nextmemaddr(m)
        ),
        pushstk(valofmemaddr(nextmemaddr(m)),s,q)
    );
xeq(rts s,m,q) =
    prog(atomofmemaddr(topstk(s,q)),popstk(s,q));
xeq(open(s),m,q) =
    prog(
        nextmemaddr(m),
        openfile(
```

```
                    atomofstr.char(topstk(s,popstk(s,popstk(s,popstk(s,q)) )) ),
                    atomoffile(topstk(s,popstk(s,popstk(s,q)) )),
                    atomofint(topstk(s,popstk(s,q)) ),
                    atomofint(topstk(s,q)),
                    popstk(s,q)
                )
            );
        xeq(close(s),m,q) =
            prog(
                nextmemaddr(m),
                closefile(
                    atomoffile(topstk(s,q)),
                    popstk(s,q)
                )
            );
        xeq(read(s),m,q) =
            prog(
                nextmemaddr(m),
                storem(
                    infile(
                        atomoffile(topstk(s,popstk(s,q)) ),
                        popstk(s,q)
                    ),
                    atomofmemaddr(topstk(s,q)),
                    popstk(s,q)
                )
            );
        xeq(write(s),m,q) =
            prog(
                nextmemaddr(m),
                outfile(
                    fetchm(
                        atomofmemaddr(topstk(s,popstk(s,q)) ),
                        popstk(s,q)
                    ),
                    atomoffile(topstk(s,q)),
                    popstk(s,q)
                )
            );
    end extend;
end am;
```

159

APPENDIX C: A SIMPLE ASSEMBLER FOR AM

1. Introduction

AMASM is an assembler which produces a relocatable load module for AM, an abstract machine interpreter. This document was adapted from Yurchak (1984), Appendix C, and constitutes the reference manual for Version 2.0-Z100. It provides a description of the syntax and semantics of the assembler as well as a description of the salient features of the AM machine and a definition of the opcodes executed by AM.

AMASM is, to the extent possible, written in portable C. The parser and scanner were produced using the Unix YACC and LEX utilities. The output from these utilities require several patches to allow compilation on the Z100 using Lattice 'C'. Readers desiring to port the code to other machines may have to make slight changes to "defines". In this implementation, **longs** are assumed to occupy 32 bits, both **int** and **short** - 16 bits, and **char** - 8 unsigned bits. Note: if the **int** size changes, then the infile and outfile functions in amstate.c must be changed.

The input syntax of AMASM is similar to that of other assemblers. It supports symbolic addresses and constants and a typical set of directives, but has no macro capabilities. The assembler accepts an ASCII source file created on a conventional text editor and produces an output file containing relocation information and AM opcodes. Invoking AM causes the output file "a.am" to be loaded and executed.

2. Differences from Version 1.0

The primary difference between AM (version 2.0-Z100) and AM (version 1.0-Z100) and their corresponding AMASM's, is the bit-mapped color display resource extension. The monitor commands are fully integrated into the AM instruction set. The form (image) data type is provided along with other supporting data types. A full set of operators allow display images to be directly manipulated.

3. Usage

AMASM is invoked with the following command line syntax:

amasm [-t] [-x] [-s] [-l] *file* ...

AMASM produces a single load module "a.am", which forms the input to the AM loader. The optional "-t" switch sends a debugging trace to "stdout" , the "-x" switch provides an extended version of the trace, and the "-s" switch provides trace of the recognized scanner tokens. The optional "-l" switch generates the listing and cross-reference file "a.x". Appended to this file is a hex dump of "a.am".

160

4. Lexical Conventions

Assembler tokens include identifiers (alternatively, "symbols" or "names"), literal constants, operators and delimiters.

4.1.. Identifiers

Legal identifiers are described by the following regular expression:

[A-Za-z_][A-Za-z0-9_]*

Identifiers consist of a letter or underline "_" followed by a string of zero or more letters, decimal digits and underlines. Upper and lower case are distinct. Identifiers may represent symbolic constants, instruction mnemonics, labels, addresses and type names.

4.2. Operators

The following are considered to be operators:

```
== != < <= > >=
+ - * / % & |
```

The meaning of the above symbols varies with context.

4.3. Literal Constants

Decimal and hexadecimal constants are described by the following regular expressions respectively:

[-+][0-9]+ | [0-9]+
$[0-9A-Fa-f]+

Decimal constants consist of an optional sign followed immediately by one or more decimal digits. Hexadecimal constants consist of the character "$" followed immediately by a string of one or more decimal digits and upper or lower case letters "A" through "F". Numeric constants may represent addresses, integer and natural numbers, boolean and character values.

Character constants consist of a single quote "'", followed either by an ASCII character, that is not a carriage return/linefeed or a numeric constant, followed by a closing single quote.

String constants consist of a string of zero or more ASCII characters (except carriage return/linefeed) enclosed in double quotes.

4.4. Blanks

Blanks and tabs are ignored by the assembler except where required to separate adjacent constants or identifiers.

4.5. Comments

The character ";" produces a comment. The assembler ignores all further characters on the line up to the terminating carriage return/linefeed.

4.6. Delimiters

All other characters found in the input stream are treated as delimiters.

5. Statements

A source program is composed of a sequence of statements, one statement per line. There are 3 kinds of statements: directives, instructions and null.

Instructions and null statements may be preceded by a label. Directives may (in some cases, must) be preceded by an identifier.

5.1. Labels & Identifiers

A label consists of an identifier followed by a colon ":". When the assembler encounters a label, the effect is to assign the current value of the location counter to the name.

An identifier preceding a directive is assigned a value whose type depends upon the directive. For instance, the **equate** directive assigns a typed value to an identifier, while the **define storage** directive assigns the current value of the location counter.

·Neither labels nor identifiers may be redefined within a single source file.

5.2. Null Statements

A null statement is an empty statement. Although ignored by the assembler, null statements may be preceded by a label.

5.3. Directive Statements

A directive is a command to the assembler to perform some sort of operation which does not involve emitting an executable instruction. Typical directives (also known as "pseudo ops" or "pseudo instructions") allocate storage for variables, make names within the current module visible to other modules and set the location counter. Directives also produce instructions for the AM linker and loader.

Directives consist of a keyword followed by zero or more arguments, depending upon the context. Directives and their syntax are described in more detail in Section 12.

5.4. Instruction Statements

Instruction statements produce the code which is ultimately executed by AM. An instruction may be preceded by a label, and consists of a keyword followed by zero or more arguments, depending upon context.

The AM instruction set and its syntax will be described in detail in Section 14.

6. The Machine

Because AM differs from conventional machines in a number of important ways, some discussion is necessary before introducing the instruction set. Outwardly similar to a number of well known examples, AM instructions form an unconventional set of primitive operations which implement a formally specified semantics. The reasons for this are described below.

AM uses a tagged architecture. Thus, each data element contains, within it, information which uniquely identifies a finite set of legal operations which may be performed upon it, as well as a range of legal values it may take on. This set of

operations and values is known formally as a **data type**. AM supports a number of data types. An element of a particular data type will be referred to throughout the rest of this manual as an **atom**.

AM physical resources are partitioned into **segments**. There are several types of segments, and these together form a conventional overall model of the familiar stored program computer. There are memory segments (primary storage), register segments (high-speed memory), display register segments (bit-mapped display memory), stacks, a monitor (display terminal attributes) and file segments (secondary storage). Segments are further partitioned into discrete, addressable elements (alternatively, "cells") which will contain atoms during the execution of a program. These elements will be referred to repeatedly as **typed values**. The reason for the distinction between atoms and values will become more clear shortly.

AM is the finite implementation of a formal specification. As such, data elements and the operations which can be applied to them must reflect a mathematical consistency not required by conventional architectures. Since all operations which affect the state of the machine must be able to "communicate" with each other during the execution of a AM program, they must do so using a common object. This object is a value. The memory, registers, display registers, stack, and files all hold values. Store, fetch, execute, read, write -- any operations which change the state of the machine -- all operate on values (i.e., storage cells). All other operations, such as "add", "multiply", "and", "or", work on atoms. Atomic operations in AM correspond to those which take place in the temporary registers of the arithmetic and logic unit of a conventional processor.

6.1. Configuration

A unique feature of AM is the ease with which it is possible to reconfigure the machine by partitioning the physical resources in different ways. A typical configuration would be something like this:

    2 memory segments
    1 register segment (with a useful number of registers)
    1 display register segment (with one or two registers)
    1 stack
    1 monitor (only one is permitted)
    16 files

The configuration chosen should provide a good indication of the types of programs AM is intended to execute.

Note that, in conventional machines, stacks are implemented in primary storage. This constitutes an overloading of data structures which obscures the intent of the user of these structures. It also creates a semantic nightmare for the specification writer. In AM, stacks take their rightful places as separate entities with easy to understand properties.

In addition to the resources listed above, AM has a conventional program counter.

6.1.1. Memory

AM memory is partitioned into segments which may be of unequal but fixed length. A program and its data will reside in memory segments. It is not necessary that code and data share the same segment, nor is it required that code and data be contiguous. The loader will determine from the **origin** directive where to load code and data values.

The AM heap is implemented as a set of operations which allocate and deallocate memory segments.

AM has a rich set of addressing modes which interact with a powerful move instruction which allows the programmer to move a value from "anywhere to anywhere".

6.1.2. Registers

AM registers form the high-speed storage into which operands are placed.

All atomic operations, such as add, divide and poffst, require operands to be in registers. Form operations are an exception. Their operands may be in either a register or a display register.

6.1.3. Display Registers

The form is the atomic data type that represents an image. Like any other atomic data type, it may be placed in any memory, register, stack or file cell. A form can not be "viewed" by the monitor unless it is in a display register.

Display registers may only contain form values. Each display register has its own window which is fixed in size but with a variable origin. The display window determines what part of the form is "viewed" by the monitor.

In general, display registers may be partitioned into multiple segments. However, the hardware on most machines will only support one segment of one or two registers. A segment of two display registers is equivalent to the idea of a "front" and "back" plane.

6.1.4. Monitor

The monitor represents a set of terminal attributes which are part of the "state of the machine". The attributes: vertical and horizontal number of pixels. vertical and horizontal screen dimensions, intensity capability and color planes are fixed for any terminal. The background color and display register selection attributes are programmable.

6.1.5. Stack

The AM stack is conventional in every respect except that it is impossible to access any value except the top. Thus, frames are implemented on the heap, not the stack.

AM has a typical set of push and pop instructions for operating on stacks.

6.1.6. Files

Input/output is implemented rather arbitrarily along the lines of system calls to an operating system and should not be considered part of AM itself.

Instructions are provided to open. close. read to and write from a file.

7. Atoms

An atom is a component of a data type. The assembler recognizes the following types of atoms:

boolean
natural
integer
character
string
intensity
color
point
rectangle
form
font
ptblt
memory address
register address
display register address
monitor attribute
stack address
file address

As operands to instruction mnemonics, these atoms form the familiar set of literal and symbolic constants found in typical assembly language programs.

With certain exceptions, atoms may appear in the form of literal constants:

100
$d0f1
´a´
"this is a string atom"

They may also appear as symbols which take on the value of the atom in some other part of the source program. With few exceptions, anywhere a literal constant may be used, a symbolic constant of the appropriate type may also be used.

The assembler distinguishes between types of atoms using syntax and context. The syntax is described below.

7.1. Boolean

A boolean atom has only two values, *true* and *false*. These values are represented to the assembler by the decimal or hexadecimal constants for 1 and 0, respectively.

0
1
$1
$0

are legal boolean atoms.

7.2. Natural

This type represents, as the name implies, the natural (unsigned) numbers. Legal values range from zero to positive infinity. Natural numbers are represented to the assembler as decimal or hexadecimal constants whose values are greater than or equal to zero.

    0
    $2f5
    240

are legal natural atoms.

7.3. Integer

Integers range from negative to positive infinity, and are specified as hexadecimal or signed or unsigned decimal constants.

    --250
    0
    $ed67f
    +10

are legal integer atoms.

7.4. Character

Character atoms may take values defined by the ASCII character set. They are represented to the assembler as literal character constants.

    ´a´
    ´r´

are legal character atoms.

7.5. String

String atoms are composed of zero or more concatenated ASCII characters. They are specified as literal strings.

    "this is a legal string atom"
    ""

are both legal string atoms.

7.6. Intensity

An intensity atom ranges from 0 to 199 decimal. It is represented as a unsigned decimal or hexadecimal constant preceded with the character " ˆ ". "@" represents the null intensity which is used to construct the null color.

    ˆ@
    ˆ0
    ˆ89
    ˆ199

are legal intensity atoms.

7.7. Color

A color atom is a composite of a red, green and blue intensity. It is represented as an ordered triple of unsigned decimal or hexadecimal constants separated by commas ",", enclosed within parentheses "(" ")" and preceded with "#". The nullcolor provides the concept of background and transparency. It is represented as the "@" enclosed within parentheses and preceded with "#".

```
#(0,0,0)
#(70,0,190)
#(@)
```

are legal color atoms.

7.8. Point

Points are a composed of integer pairs. The x and y coordinates correspond to the first and second integers respectively. Increasing integer values represents positions shifted right and up. A point is represented as an ordered pair of decimal or hexadecimal constants separated by a comma "," and enclosed within parentheses "(" ")".

```
(0,0)
(4,1047)
(-8,25)
(-50677,-293399)
```

are legal point atoms.

7.9. Rectangles

Rectangles are composed of a pair of points which represent the opposing corners. A rectangle is represented as an unordered pair of points separated by a colon ":" and enclosed with in square brackets "[" "]".

```
[(0,0):(0,0)]
[(0,0):(50,45)]
[(50,45):(0,0)]
[(-20,-20000):(30,59)]
```

are legal rectangle atoms.

7.10. Form

A form atom is a composite structure. It has a two dimensional size and a color map which is an array of colors with each color corresponding to a point in its area. The form atom has no literal constant representation. It is created using the operator, *newfrm*, and modified using other operators.

7.11. Font

A font atom is an array of forms. The font atom has no literal constant representation. It is constructed from the operator, *newfnt*, and modified using other operators.

7.12. Ptblt

A ptblt atom is a composite of three rectangles and a natural which represents a copy rule. The ptblt atom has no literal constant representation. It is constructed from the operator, *newblt*, and is modified using other operators.

7.13. Memory Address

Memory address atoms consist of two components: a segment address, and an element address. Memory addresses are represented as an ordered pair of unsigned decimal or hexadecimal constants, separated by a colon ":" and enclosed within parentheses "(" ")".

(0:100)

represents memory segment 0, element 100.

(2:$10)

represents segment 2, element 16.

Segment and element addresses start at 0. The number and size of available memory segments depends upon the current configuration of AM.

Labels are considered memory address atoms, as are names which appear to left of the **define storage** and **define constant** directives.

7.14. Register Address

Register address atoms have a syntax identical to that of memory addresses except that a lower case "r" is prepended to the address.

r(0:3)

refers to register segment 0, register 3.

Segment and element addresses start, as with memory addresses, at 0. The number of register segments, and the number of registers within each segment, varies as determined by the current AM configuration.

7.15. Display Register Address

Display register address atoms have a syntax identical to that of register addresses except that the lower case "r" is replaced with a lower case "d".

d(0:1)

refers to display register segment 0, register 1.

Segment and element addresses start at 0. The number of display register segments, and the number of display registers within each segment, varies as determined by the current AM configuration.

7.16. Monitor Attribute

The monitor consists of eight attributes values which are:

x - represents number of horizontal pixels (natural)
y - represents number of vertical pixels (natural)
v - represents screen height in inches (natural)
h - represents screen width in inches (natural)
i - represents intensity capability (natural)

c - represents number of color planes (natural)
b - current background color (color)
d - selected display register to view
        (display register address)

A monitor attribute is represented by a dash "-" followed by one of the above characters for the indicated attribute.

```
-x
-y
-b
```

are all legal monitor attribute atoms.

7.17. Stack Address

A stack address has only one component: the segment address. Stack addresses are specified by prepending a lower case "s" to an unsigned decimal or hexadecimal constant enclosed within parentheses.

```
s(2)
```

refers to stack segment 2.

Stack addresses begin at 0. The number of stacks depends upon AM's configuration.

7.18. File Addresses

File address atoms may not appear in a program except within typed values. File address atoms are represented as unsigned integer or hexadecimal constants.

File addresses start at 0. The number of files which may be open at one time is determined by the current AM configuration. The first three file addresses (0,1,2) are normally opened automatically by AM when a program is loaded.

8. Typed Values

Some of the atomic types may also appear as typed values in certain instructions and directives. A typed (immediate) value is represented as an ordered pair consisting of a keyword representing the type, and the atom itself, separated by a comma "," and enclosed within curly braces "{""}".

```
{int,100}
```

represents the integer value 100.

```
{addr,(1:100)}
```

represents memory address value (1:100).

A list of the types which may be used as immediate values alongside the corresponding keywords appears below:

```
bool - boolean
nat - natural
int - integer
char - character
string - character string
```

169

intens - intensity
color - color
pnt - point
rct - rectangle
addr - memory address
file - file address

Immediate values are used, as in conventional assembly languages, for loading constants into cells, initializing storage, pushing parameters to subroutines on the stack, and so on.

A special syntax may be applied when expressing typed values for the **define storage** and **define constant** directives. The type keyword may be followed by a list of atoms of the appropriate type, separated by commas.

{int,1,2,3,4,5,6,7,8}

shows an example of this.

9. Expressions

An expression may be substituted anywhere an integer or natural atom is called for. The expression must be a sequence of integer/natural atoms (and symbolic constants equated to integer/natural atoms) separated by operators and grouping symbols which evaluates to an atom of the type called for where the expression is used.

9.1. Expression Operators

Legal operators are (in order of increasing precedence):

|	- or
&	- and
+ -	- addition and subtraction
* / %	- multiplication, division, and modulus
-	- unary minus

Expressions may be grouped using parentheses "(" ")".

10. Notation

Throughout the rest of this manual, the following notational conventions will be used to describe the syntax of directives and instructions.

A - atom
V - typed value
N - natural atom
I - integer atom
M - memory address atom
R - register address atom
D - display register address atom
C - either a display or a high speed register address atom
T - monitor attribute atom
S - stack address atom
< > - items enclosed within angle brackets are arguments

170

[ ] - items enclosed in square brackets are optional

<ea> - effective address

<ev> - effective value

11. Data Format

AMASM emits object code and directives using AM I/O modules. The object module is, thus, directly readable by AM. A linker and loader may be written either in a high level language, or AM assembler.

The data and object module formats described below are a direct reflection of AM's tagged architecture. The following conventions will apply:

- All numbers show are in hexadecimal.

- The letter "H" is a place holder signifying any 4-bit value.

- The letter "D" is a place holder signifying any 32-bit value.

- The letter "P" is a place holder signifying a 32-bit pointer.

- The general form of a typed value is

| tag | | val |

where "tag" is a 16-bit type field, and "val" is either an 8 to 32-bit value or a 32-bit pointer.

Note the following:

- Character string atoms and values have a 16-bit size field inserted after the type field which indicates the number of characters in the value field (including the terminating null). This size field is omitted in memory (since it is not needed) and replaced by a pointer to the string.

- Instruction values have a 32-bit pointer following the type field, which points to an array of values. The first value is the opcode followed by the operands. The number of operands is encoded in the opcode.

- Form values have a 32-bit pointer to a form header. The header contains the form's rectangle and a pointer to the cmap which is an array of colors. The length of the cmap is determined from the form's rectangle.

- Font values have a 32-bit pointer to a font header. The header contains the font's rectangle and a 128 member array of cmap pointers.

A number of the formats listed below are not described elsewhere in this manual since they are either not accessible to the programmer, or are implied by context.

11.1. Atom Formats

boolean - | 0001 | | HH |

natural - | 0002 | | HHHH |

integer - | 0003 | | HHHH |

character - | 0004 | | HH |

character string - | 0005 | | P | | HH...00 |

171

intensity - [0006] [HH]

color - [0007] [HH HH HH]

point - [0008] [P] [DD]

rectangle - [0009] [P] [DDDD]

form - [000A] [P] [DD DD] [P] [- cmap array -]

font - [000B] [P] [DD DD] [- 128 P's -]

ptblt - [000C] [P] [DDDD DDDD DDDD HH]

memory address - [0030] [D]

register address - [0031] [D]

display register address - [0032] [D]

monitor attribute - [0033] [HH]

stack address - [0034] [D]

file address - [0035] [HHHH]

monadic operator - [0040] [HHHH]

dyadic operator - [0041] [HHHH]

triadic operator - [0042] [HHHH]

quadadic operator - [0043] [HHHH]

sexadic operator - [0044] [HHHH]

octadic operator - [0045] [HHHH]

relational operator - [0046] [HHHH]

boolean comparator - [0047] [HHHH]

11.2. Value Formats

boolean - [0201] [HH]

natural - [0202] [HHHH]

integer - [0203] [HHHH]

character - [0204] [HH]

character string - [0205] [P] [HH...00]

intensity - [0206] [HH]

color - [0207] [HH HH HH]

point - [0208] [P] [DD]

rectangle - [0209] [P] [DDDD]

form - [020A] [P] [DD DD] [P] [- cmap array -]

font - [020B] [P] [DD DD] [- 128 P's -]

ptblt - [020C] [P] [DDDD DDDD DDDD HH]

memory address - [0230] [D]

register address - ☐0231☐ ☐D☐

display register address - ☐0232☐ ☐D☐

monitor attribute - ☐0233☐ ☐HH☐

stack address - ☐0234☐ ☐D☐

file address - ☐0235☐ ☐HHHH☐

instruction - ☐0250☐ ☐P☐ ☐HHHH☐ ☐zero or more operand atoms☐

11.3. Object Module Format

The structure of an object module is very simple. The only object always found is a leading **org** directive. Next, if any symbols were declared global or external in the source module, a pseudo instruction will be emitted for each such symbol. The rest of the file contains executable and pseudo instructions emitted as they occur in the source.

12. Assembler Directives

AMASM recognizes the following directives:

equ - equate
org - absolute origin
rorg - relative origin
extern - external symbol
globl - global symbol
trace - trace execution
ds - define storage
dc - define constant

Directives do not produce code which will be executed by AM, but they may cause linker/loader instructions to be emitted. The meaning and syntax of each directive is described in the following pages.

173

Syntax:

    <name> equ <equivalence>

    where:
    <name> is any legal identifier
    <equivalence> is any atom or typed value

Description:

    The symbol <name> is assigned the value of <equivalence>. Elsewhere in the source module, the symbol may be used in place of a literal value of the same type as <equivalence> using the following syntax:

- - If the symbol represents a memory address **atom**, the symbol may be used directly.

- If the symbol represents a typed (immediate) value, it must be enclosed in curly braces "{" "}".

- If the symbol represents an integer or natural atom, it must be preceded by a pound sign "#".

Example:

```
progseg       equ     (0:0)
dataseg       equ     (1:100)
offset        equ     10
datafile      equ     {file,3}

              org     progseg
              move    {addr,data},r(0:0)
              move    {int,100},r(0:0)@#offset

              push    {string,"test.dat"},s(0)
              push    {datafile}, s(0)
              push    {int,0},s(0)
              push    {int,0},s(0)
              open    s(0)
              stop

              org     dataseg
data          ds      100
```

    "progseg" and "dataseg" are equated to memory address atoms.

    "offset" is equated to the integer atom 10.

    "datafile" is equated to the file address value {file,3}.

Format:

    equ does not cause an emission.

Syntax:

　　org [M]

Description:

　　The location counter is reset to M, if specified; otherwise it remains unchanged. All memory addresses and labels specified after an **org** directive up to the next **org** or **rorg** directive not explicitly expressed as displacements are treated as absolute addresses. Code generated after an **org** directive up to the next **org** or **rorg** directive is not relocatable.

Example:

```
               org

               move  (0:0),r(0:0)

               org    (1:0)
     data      ds     {int,100},{nat,0}
```

Format:

　　 0250 1801 0230 D

Syntax:

    rorg [M]

Description:

    The location counter is reset to M, if specified; otherwise it remains unchanged. All memory addresses and labels specified after a **rorg** directive up to the next **org** or **rorg** directive are computed as displacements. Code generated after a **rorg** directive up to the next **org** or **rorg** directive is relocatable (program counter independent).

Example:

```
            rorg

            move    {int,100},data
            jsr     stuff
            stop

data        ds      10
```

In the above example, the **move** would be emitted using destination program counter relative addressing.

Format:

   0250   1801   0230   D

Syntax:

extern <name>...

where:
<name> is any legal identifier

Description:

The list of symbols is made visible to the current module and are assumed to be defined elsewhere. An error is flagged if a symbol in the list is not referenced somewhere within the current module. It is also an error for any symbol in the list to be defined within the current module.

Example:

extern expon

push   {int,100},s(0)
jsr   expon,s(0)

Format:

For each symbol declared external, an **extern** pseudo op is emitted, followed by a string containing the symbol.

| 0250 | 1802 | 0205 | P | HH...00 |

Syntax:

    globl <name>...

    where:
    <name> is any legal identifier

Description:

    The list of symbols is made visible to external modules. Each <name> in the list must be defined as a memory address somewhere within the current module.

Example:

              globl   test,data

    test:
              move (0:0),r(0:0)
              stop
    data   ds   10

    "test" and "data" are made visible to other modules.

Format:

    For each symbol declared global, a **globl** pseudo op is emitted, followed by a string containing the symbol, followed by a memory address representing the value of the symbol.

    $\boxed{0250}$ $\boxed{1803}$ $\boxed{0005}$ $\boxed{\text{P}}$ $\boxed{\text{HH...00}}$ $\boxed{0230}$ $\boxed{\text{D}}$

Syntax:

    trace <flag>,<toggle>

    where:
    <flag> is "-t" for normal trace and "-x" is for extended trace
    <toggle> is "+" for on and "-" for off

Description:

A trace of the programs execution is available in two modes, normal and extended. The normal mode traces the main function calls and the major paths through them. The extend mode include the normal trace plus memory allocation calls and creation of temporary values. The trace directive may be selected in the command line when AM is invoked, or embedded in the source code to enable trace over selected portions of the program.

Example:

    progseg   equ   (0:0)
                      org   progseg
                      move   {addr,data},r(0:0)
                      trace -t,+
                      move   {int,100},r(0:0)@
                      trace -t,-
                      push   {int,0},s(0)
                      stop
    data   ds . 100

Format:

| 0250 | 3800 | 0204 | HH | 0203 | HHHH |

Syntax:

    [<name>] ds N [V...]
    [<name>] ds [N] V...

    where:
    <name> is an optional identifier

    ds permits a list of atoms to follow the type keyword of each value.

Description:

ds allocates storage for values starting at the current value of the location counter.

- If N is specified and N is greater than or equal to the number of values in the list, space for N values is allocated and the location counter is incremented by N.

- If N is specified and N is less than the number of values in the list, N is ignored.

- If N is not specified, the amount of storage allocated is equal to the number of values in the list. The location counter is incremented by this number.

- If a value list is specified, the allocated cells will be initialized to those values, beginning with the first.

- Cells allocated but not initialized are considered to hold undefined values. It is an error to attempt to read an undefined value.

Example:

```
data1        ds     10
data2        ds     10 {int,100},{nat,0,20,40}
data3        ds     {char,'a','b'}
             ds     {string,"this is a sting value"}
```

The first ds allocates 10 values and leaves them undefined. "data1" may be used to index into those values.

The second also allocates 10 values, but initializes the first to the integer 100, and the next 3 to the naturals 0, 20, and 40. The last 6 values are left undefined.

The third ds shown allocates 2 character values.

The fourth allocates a single string value. No identifier was specified.

Format:

A typed value is emitted for each value in the list. In addition, ds will emit an org pseudo op (see org) whenever the number of values in the value list is less than N.

Syntax:

   [<name>] dc V...

   where:

   <name> is an optional identifier

   dc permits a list of atoms to follow the type keyword of each value.

Description:

   dc allocates and initializes storage from a list of values starting at the current value of the location counter.

Example:

   data3   dc   {char,'a','b'}
                  dc   {string,"this is a string value"}

   The first **ds** shown allocates 2 character values.

   The second allocates a single string value. No identifier was specified.

Format:

   A typed value is emitted for each value in the list.

13. Addressing Modes

AM supports 11 addressing modes:

d - display register direct
r - register direct
ri - register indirect
rid - register indirect with displacement
ridn - n-level register indirect with displacement
m - memory absolute
mi - memory indirect
pcr - program counter relative
i - immediate value
a - immediate atom
s - stack direct

Like other more familiar processors, not all AM instructions can use all of the addressing modes.

In addition, AMASM supports address expressions, which provides a rudimentary indexing capability.

13.1. Display Register Direct

The form operand is in the display register.

Syntax: D

Format:

| 0232 | D |

13.2. Register Direct

The operand is in a register.

Syntax: R

Format:

| 0231 | D |

13.3. Register Indirect

The address of the operand is in a register.

Syntax: R@

R   - holds the operand address

Format:

| 0231 | D |

13.4. Register Indirect with Displacement

The address of the operand is the sum of the address in a register and an integer displacement.

Syntax: R@I

R   - holds a base address

I   - an integer displacement

Format:

| 0231 | D | 0203 | HHHH | .

13.5. N-level Register Indirect with Displacement
The address of the operand is the sum of the address obtained from the nth link in a chain of dynamic links and an integer displacement.

Syntax: RN@I

R   - holds the current frame pointer
N   - a non-negative frame reference
I   - an integer frame displacement

(RN@I is equivalent to R@I)

Format:

| 0231 | D | 0202 | HHHH | 0203 | HHHH |

13.6. Memory Absolute
Syntax: M

M   - the operand address

Format:

| 0230 | D |

13.7. Memory Indirect
The address of the operand is in a memory cell.

Syntax: M@

M   - a pointer to the operand address

Format:

| 0230 | D |

13.8. Program Counter Relative
The address of the operand is the sum of the program counter and an integer displacement.

Syntax: M

M   - the operand address

The specified address must be in the same module as the instruction. The assembler automatically computes the displacement. Program counter relative is specified for a block by placing a **rorg** directive at the top of the block.

Format:

| 0203 | D |

184

13.9. Immediate Value
The operand is an immediate value.

Syntax: V

    V   - any typed value

Format:

tag		val

13.10. Immediate Atom
The operand is an atom.

Syntax: A

    A   - usually an integer or natural

Format:

tag		val

13.11. Stack Direct
The operand is a stack.

Syntax: S

Format:

0234	D

14. Instruction Set
The AM instruction set is simple but powerful. The rigid data types make it meaningless to specify operations like shift and mask, thus removing some of the programmer's freedom to muck with data in arbitrary ways. The tagged architecture will detect errors like jumping to data, or accessing instructions as data, as well as the more common bounds checking performed by runtime libraries.

14.1. Machine Errors
The following errors are detected by AM during loading and execution:

- attempt to execute a non-instruction
- attempt to execute an illegal instruction
- memory segment not defined
- memory segment overflow
- memory segment underflow
- register segment not defined
- register segment underflow
- register segment underflow
- display register segment not defined
- stack segment not defined
- undefined monitor attribute
- <file> contains unresolved references

- attempt to convert negative int to nat
- no predecessor to zeronat
- no predecessor to minintens
- no successor to maxintens
- addition illegal with nullintens
- subtraction illegal with nullintens
- gtintens illegal with nullintens
- ltintens illegal with nullintens
- geintens illegal with nullintens
- leintens illegal with nullintens
- illegal color definition
- form is not correct size for font
- icon is undefined
- unknown operator to applymop
- unknown operator to applydop
- unknown operator to applytop
- unknown operator to applyqop
- unknown operator to applysop
- unknown operator to applyoop
- unknown operator to applyrop
- unknown operator to applybop
- type error - GT
- type error - GE
- type error - LT
- type error - LE
- no more segment available
- attempt to free invalid memory segment
- attempt to free non-allocated segment
- stack empty
- stack overflow
- stack underflow
- file already open
- unable to close file
- unable to open <file>
- file already closed
- file not open
- file not open for reading
- file not open for writing
- reading file, type not recognized
- error reading file
- writing file, type not recognized
- invalid memory segment
- memory segment not allocated
- invalid memory address
- invalid register segment
- invalid register address

- invalid stack segment
- invalid file descriptor
- attempt to return head of null string
- value not of type bool
- atom not of type bool
- value not of type int
- atom not of type int
- value not of type nat
- atom not of type nat
- value not of type char
- atom not of type char
- value not of type string
- atom not of type string
- value not of type ilev
- atom not of type ilev
- value not of type colr
- atom not of type colr
- value not of type pnt
- atom not of type pnt
- value not of type rct
- atom not of type rct
- value not of type form
- atom not of type form
- value not of type font
- atom not of type font
- value not of type ptblt
- atom not of type ptblt
- value not of type mad
- atom not of type mad
- value not of type rad
- atom not of type rad
- value not of type dad
- atom not of type dad
- value not of type mattribute
- atom not of type mattribute
- value not of type sad
- atom not of type sad
- value not of type file
- atom not of type file
- value not of type mop
- atom not of type mop
- value not of type dop
- atom not of type dop
- value not of type top
- atom not of type top
- value not of type qop

- atom not of type qop
- value not of type sop
- atom not of type sop
- value not of type oop
- atom not of type oop
- value not of type rop
- atom not of type rop
- value not of type bop
- atom not of type bop
- value not of type instr
- atom not of type instr
- type error

All machine errors are fatal.

14.2. Assembler Errors

AMASM will detect and report the following errors:

- symbol not an address
- symbol defined locally
- <symbol> does not match declared type
- relative memory indirect not permitted
- symbol not a value
- symbol not an integer
- intensity value exceeds range
- symbols declared but not referenced
- displacement from external addresses not permitted
- relative addressing not permitted between segments
- out of symbol space
- symbol declared external
- symbol already defined
- symbol not of same type
- impossible value for given type
- syntax error

Assembler errors are not fatal, but will prevent the creation of the object module and, usually, the cross-reference file.

14.3. AM Operations

AM supports a useful set of monadic, dyadic, triadic, quadadic, sexadic, octadic, relational and test operators. These operators are to be used with the monad, dyad, triad, quad, sexad, octad, if and test instructions. The mnemonics/symbols for each operator along with the data types to which each may be applied are described below.

14.3.1. Monadic Operators (MOP's)

not - boolean negation

> **not** accepts a boolean argument and returns its negation

abs - absolute value

> **abs** accepts an integer argument and returns is absolute value

ntoi - natural to integer

> **ntoi** accepts a natural argument and converts it to an integer

iton - integer to natural

> **iton** accepts an integer argument and converts it to a natural

len - string length

> **len** accepts a string argument and returns its length as a natural number.

make - make a string

> This operator accepts a character argument and returns a string of length 1.

head - the head of a string

> This operator accepts a string argument and returns the character at its head. It is an error to take the head of an empty string.

tail - the rest of a string

> **tail** accepts a string argument and returns a string containing all but the first character. The tail of an empty string is the empty string.

rcmp,gcmp,bcmp - color components

> **rcmp**, **gcmp** and **bcmp** accept a color argument and return the respective red, green, or blue component of the color.

xcord,ycord - point coordinate

> **xcord** and **ycord** accept a point argument and return the respective coordinate integer.

origin,corner - rectangle corner points

> These operators accept a rectangle argument and return a corner point. **Origin** returns the lower left and **corner** the upper right.

xdim,ydim - rectangle dimensions

> **xdim** and **ydim** accept a rectangle argument and return the respective dimension integer.

newfrm - new form

newfrm accepts a rectangle argument and returns a new blank form whose rectangle is the same as the input rectangle.

farea - form area

farea accepts a form argument and returns its rectangle.

gblts,gbltd,gbltc - get ptblt rectangles

These operators accept a ptblt argument and return the specified rectangle. **gblts** returns the source, **gbltd** returns the destination, and **gbltc** returns the clipping rectangle

gbltr - get ptblt rule

· **gbltr** accepts a ptblt argument and returns the natural that represents the copy rule.

newfnt - new font

newfnt accepts a rectangle argument and returns an empty font whose icon rectangles are the same as the input rectangle.

rctfnt - rectange of font

rctfnt accepts a font argument and returns its rectangle.

lenfnt - length of font

lenfnt accepts a font argument and returns the number of icons in it as a natural.

14.3.2. Dyadic Operators (DOP´s)

and,or

and and **or** accept two boolean arguments and return a boolean result.

add,sub,mul,div,mod - computational operators

These operators accept integer, natural or intensity arguments (both of the same type) and return a result of that type. Divide by zero returns an error. **div** discards any remainder. **mod** returns the remainder. **mul**, **div** and **mod** do not apply to intensity arguments.

cat - string concatenation

cat accepts two string arguments and returns the concatenation of the first onto the second.

loc - point location

loc accepts two integer arguments and returns the defined point.

Usage - loc(x,y) where x is the x coordinate integer and y is the y coordinate integer.

area - rectangle definition

area accepts two unordered point arguments and returns the defined rectangle.

inrct - point in rectangle

inrct accepts a point and a rectangle argument, checks if the point is inside the area of the rectangle, and returns the boolean result.

Usage - inrct(p,r) where p is a point and r is a rectangle.

intrct - rectangle intersection

intrct accepts two rectangle arguments and returns the intersection rectangle.

putrct - put rectangle at

putrct accepts a point and a rectangle argument and returns the rectangle with the same area as the input and its origin at the point argument.

Usage - putrct(p,r) where p is a point and r is a rectangle.

mapsp,mapps - conversion operators

These operators convert points between point coordinates and font spot coordinates. They accept a point and a font argument and return a point. **mapsp** takes a spot coordinate and based on the font size returns its origin point, e. g., the origin point of spot (2,3) for a 10 by 10 font is point (20,30). **mapps** takes a point and returns the font spot that it falls inside, e. g., the point (21,31) for a 10 by 10 font is in spot (2,3).

Usage:

- mapsp(f,p) where f is a font and p is a point,

- mapps(f,p) where f is a font and p is a point.

gcolor - get color

gcolor accepts a point and a form argument and returns the form's color at that point. bp

Usage - gcolor(p,f) where p is a point and f is a font.

fill - fill the form

fill accepts a color and a form and returns the form with all its points set to the color argument.

Usage - fill(c,f) where c is a color and f is a font.

sblts,sbltd,sbltc - set ptblt rectangles

These operators accept a rectangle and a ptblt argument and return the ptblt with the specified rectangle set to the rectangle argument.

sblts sets the source, **sbltd** sets the destination, and **sbltc** sets the clipping rectangle

Usage - sblt_(r,b) where r is a rectangle and b is ptblt.

sbltr - set ptblt rule

sbltr accepts a natural and a ptblt argument and returns the ptblt with copy rule set to the natural argument.

Usage - sbltr(n,b) where n is a natural and b is ptblt.

infnt - is icon in font

infnt accepts a natural and a font argument and returns a boolean result based on whether the icon indexed by the natural argument is defined.

Usage - infnt(n,f) where n is a natural and f is a font.

dfnt - delete icon

dfnt accepts a natural and a font argument and returns the font with the indexed icon deleted.

Usage - dfnt(n,f) where n is a natural and f is a font.

gfnt - get icon

gfnt accepts a natural and a font argument and returns the form of the icon indexed.

Usage - gfnt(n,f) where n is a natural and f is a font.

14.3.3. Triadic Operators (TOP's)

dcolor - define color

dcolor accepts three intensity arguments and returns the defined color.

Usage - dcolor(r,g,b) where r is the red intensity, g is the green intensity, and b is the blue intensity.

poffst

poffst accepts a point and two integer arguments and returns the point that is offset from the point argument by the integer arguments.

Usage - poffst(x,y,p) where x and y are the offset integers and p is the reference point.

sftrct - shift rectangle

sftrct accepts a rectangle and two integer arguments and returns the rectangle formed by offsetting its origin by the integer arguments.

Usage - sftrct(x,y,r) where x and y are the offset integers and r is the reference rectangle.

scolor - set color

scolor accepts a color, a point and a form argument and returns the form with its point argument set to the color argument.

Usage - scolor(p,c,f) where p is the point, c is the color, and f is the font.

invfrm - inverse form

invfrm accepts a form and two color arguments and returns the form with its fore and background colors inversed by the color arguments.

Usage - invfrm(fg,bg,frm) where fg is the new foreground color, bg is the new background color, and frm is the form to be inversed.

sfnt - set font

sfnt accepts a natural, a form, and a font and returns the font with the new icon inserted that is defined by the form and natural arguments.

Usage - sfnt(frm,n,fnt) where frm is the icon form, n is the index, and fnt is the font.

14.3.4. Quadadic Operators (QOP's)

foffst - font offset

foffst accepts two integer arguments as an offset, a point argument and a font argument. It returns the spot origin point based on the spot coordinate offset from the point argument, e. g., a font size of 10 by 10 which is offset 2,3 from point (5,5) returns the spot origin point at (25,35).

Usage - foffst(x,y,fnt,p) where x and y are the offset integers, fnt is the basis font, and, p is the reference point.

cpfrm - form copyblt

cpfrm merges a source and a mask form with a destination form using the parameters in ptblt. It accepts a ptblt and three form arguments and returns the resultant form.

Usage - cpfrm(pb,s,m,d) where pb is the governing ptblt, s is the source form, m is the mask form, and d is the destination form.

14.3.5. Sexadic Operators (SOP´s)

drawln - draw line

> **drawln** draws a line from point y to point z on the destination form, using the specified brush and mask forms. It accepts two point arguments, three form arguments and a ptblt argument and returns a form.
>
> **Usage** - drawln(x,y,pb,b,m,d) where y is the start point, z is the end point, pb is the ptblt, b is the brush form, m is the mask form, and d is the destination form.

cpfnt - copy font

> **cpfnt** copies a font icon to a designated point on the destination form. It accepts a natural and a font argument which defines the source form, a point argument for the target location, two form arguments and a ptblt argument and returns the resultant form.
>
> **Usage** - cpfnt(p,pb,n,fnt,m,d) where p is the target location, pb is the ptblt, n is the font index, fnt is the font, m is the mask form, and d is the destination form,

14.3.6. Octadic Operators (OOP´s)

invfnt - inverse font

> **invfnt** performs the same operation as cpfnt except that the font icon is combined with inverse coloring. It accepts a the same arguments plus two color arguments and returns the resultant form.
>
> **Usage** - invfnt(fg,bg,p,pb,n,fnt,m,d) where fg is the new foreground color, bg is the new background color, p is the target location, pb is the ptblt, n is the font index, fnt is the font, m is the mask form, and d is the destination form,

14.3.7. Relational Operators (ROP´s)

The relational operators are:

```
==      - equality
>       - greater than
>=      - greater than or equal to
<       - less than
<=      - less than or equal to
!=      - not equal to
```

They may be applied to int, nat, char, string, intens, and pnt.

If == or != are applied to arguments of different types, == returns false, != returns true. This applies also to types not listed above. >,>=,< and <= return an error if their arguments are not of the same type.

Relational operators return a boolean result.

14.3.8. Test Operators (BOP's)

These operators permit the programmer to test a cell for type before attempting to access it. These are necessary because AM considers it a fatal error to read from an undefined cell or apply an operator of one type on data of another. The test operators are the same as the type mnemonics, plus a mnemonic for testing undefined values:

bool
nat
int
char
string
intens
color
pnt
rct
form
font
ptblt
instr
addr
file
undef

Test operators accept a typed value and return true if the value is of the specified type, false otherwise. **undef** returns true if a value is undefined, false otherwise.

Syntax:

    offset I,R

    R must contain a memory address value

Operation:

    $R + I \rightarrow R$

Description:

    The sum of I and the address in R is stored in R.

Example:

            offset   20,r(0:0)

Addressing Modes:

    I: a

    R: r

Format:

    | 0250 | P | 3810 | operands |

Syntax:

    link R,N

Operation:

    R@ --> address@
    address --> R

Description:

    A segment of N cells is allocated from the heap. The value stored in R is save at the base address of the segment. The segment base address is returned in R.

    This instruction is designed to create dynamic links for local environments.

Example:

```
proc:           link    r(0:5),1
                move    r(0:5)2@4,r(0:0)
                add     {int,100},r(0:0)
                move    r(0:0),r(0:5)2@4
                unlink r(0:5)
                rts
```

    Above is an example of uplevel addressing.

Addressing Modes:

    R: r
    N: a

Format:

    | 0250 | | P | | 3811 | | operands |

Syntax:

 unlink R

Operation:

 R@ --> R

Description:

 The value in the base address of the segment pointed to by R is returned in R. The segment is freed.

Example:

```
proc:        link    r(0:5),1
             move    r(0:5)2@4,r(0:0)
             add     {int,100},r(0:0)
             move    r(0:0),r(0:5)2@4
             unlink  r(0:5)
             rts
```

Addressing Modes:

 R: r

Format:

 | 0250 | | P | | 2812 | | operand |

Syntax:

    gdwin D,R

Operation:

    D --> R

Description:

    The value of the display window origin point at D is stored in R.

Example:

                gdwin d(0:0),r(0:0)

Addressing Modes:

    R: r

Format:

    | 0250 | | P | | 3813 | | operands |

Syntax:

sdwin R,D

R must contain a point value

Operation:

R --> D

Description:

The display window origin point at D is set to the point value in R.

Example:

sdwin r(0:0),d(0:0)

Addressing Modes:

R: r

Format:

| 0250 | P | 3814 | operands |

Syntax:

gmtr T,R

Operation:

T --> R

Description:

The T value is stored in R.

Example:

gmtr   -b,r(0:0)

Addressing Modes:

R: r

Format:

| 0250 | | P | { | 2815 |...| 281C | } | operand |

Syntax:

smtr R,T

R must contain a value appropriate for the selected attribute.

Operation:

R --> T

Description:

The T value is set to the value in R

Example:

smtr   r(0:0),-d

Addressing Modes:

R: r

Format:

0250 P { 281D ... 2824 } operand

Syntax:

<mop> C

where:
<mop> is a monadic operator

Operation:

<mop> C --> C

Description:

The operator corresponding to <mop> is applied to C and the result stored in C.

Example:

not   r(0:0)

Addressing Modes:

C: r,d

Format:

| 0250 | P | 3830 | operand |

Syntax:

    <mop> Cx,Cy

    where:
    <mop> is a monadic operator

Operation:

    <mop> Cx --> Cy

Description:

    The operator corresponding to <mop> is applied to Cx and the result stored in Cy.

Example:

              not   r(0:0),r(1:0)
              farea   d(0:0),r(0:0)

Addressing Modes:

    Cx: r,d

    Cy: r,d

Format:

    | 0250 | | P | | 4831 | | operands |

Syntax:

    <mop> V,C

    where:
    <mop> is a monadic operator

Operation:

    <mop> V --> C

Description:

    The operator corresponding to <mop> is applied to the immediate value V and the result stored in C.

Example:

        not   {addr,flag},r(1:0)
        newfrm   {addr,rctsize},d(0:0)

Addressing Modes:

    V: i

    C: r,d

Format:

    | 0250 | | P | | 4832 | | operands |

Syntax:

    <dop> Cx,Cy

    where:
    <dop> is a dyadic operator

Operation:

    Cy <dop> Cx --> Cy

Description:

    The operation corresponding to <dop> is applied to the operands and the result stored in Cy.

Example:

        and   r(0:0),r(0:1)
        fill   r(0:0),d(0:0)

Addressing Modes:

    Cx: r,d

    Cy: r,d

Format:

   | 0250 | P | 4833 | operands |

Syntax:

 <dop> V,C

 where:
 <dop> is a dyadic operator

Operation:

 C <dop> V --> C

Description:

 The operation corresponding to <dop> is applied to the operands and the result stored in C.

Example:

    sub {int,100},r(0:1)
    fill {color,#(10,10,10)},d(0:0)

Addressing Modes:

 V: i

 C: r,d

Format:

 | 0250 | | P | | 4834 | | operands |

Syntax:

    <dop> Cx,Cy,Cz

    where:
    <dop> is a dyadic operator

Operation:

    Cy <dop> Cx --> Cz

Description:

    The operation corresponding to <dop> is applied to Cx and Cy and the result stored in Cz.

Example:

                add   r(0:0),r(0:1),r(0:3)
                gcolor r(0:0),d(0:0),r(0:1)

    <dop> Cx,Cy,Cy is equivalent to <dop> Cx,Cy

Addressing Modes:

    Cx: r,d
    Cy: r,d
    Cz: r,d

Format:

    | 0250 | | P | | 4835 | | operands |

Syntax:

    <dop> V,Cx,Cy

    where:
    <dop> is a dyadic operator

Operation:

    Cx <dop> V --> Cy

Description:

    The operation corresponding to <dop> is applied to V and Cx and the result stored in Cy.

Example:

        add   {int,100},r(0:0),r(0:1)
        gcolor {pnt,(0,0)},d(0:0),r(0:0)

    <dop> V,Cx,Cx is equivalent to <dop> V,Cx

Addressing Modes:

    V: i

    Cx: r,d

    Cy: r,d

Format:

    | 0250 | | P | | 4836 | | operands |

Syntax:

<top> Cx,Cy,Cz

where:
<top> is a triadic operator

Operation:

<top> Cx,Cy,Cz --> Cz

Description:

The operation corresponding to <top> is applied to the operands and the result stored in Cz.

Example:

sfnt   r(0:0),r(0:1),r(0:2)

Addressing Modes:

Cx: r,d
Cy: r,d
Cz: r,d

Format:

| 0250 | P | 4837 | operands |

Syntax:

    <top> Cw,Cx,Cy,Cz

    where:
    <top> is a triadic operator

Operation:

    <top> Cw,Cx,Cy --> Cz

Description:

    The operation corresponding to <top> is applied to the operands and the result stored in Cz.

Example:

             scolor r(0:0),r(0:1),d(0:2),r(0:3)

Addressing Modes:

    Cw: r,d
    Cx: r,d
    Cy: r,d
    Cz: r,d

Format:

    | 0250 | | P | | 4838 | | operands |

Syntax:

   <qop> Cw,Cx,Cy,Cz

   where:
   <qop> is a quadadic operator

Operation:

   <qop> Cw,Cx,Cy,Cz --> Cz

Description:

   The operation corresponding to <qop> is applied to the operands and the result stored in Cz.

Example:

   cpfrm r(0:0),d(0:0),r(0:1),d(0:1)

Addressing Modes:

   Cw: r,d

   Cx: r,d

   Cy: r,d

   Cz: r,d

Format:

| 0250 | P | 4839 | operands |

Syntax:

   <qop> Cv,Cw,Cx,Cy,Cz

   where:
   <qop> is a quadadic operator

Operation:

   <qop> Cv,Cw,Cx,Cy --> Cz

Description:

   The operation corresponding to <qop> is applied to the operands and the result stored in Cz.

Example:

   cpfrm r(0:0),r(0:1),r(0:2),r(0:3),d(0:0)

Addressing Modes:

   Cv: r,d
   Cw: r,d
   Cx: r,d
   Cy: r,d
   Cz: r,d

Format:

   | 0250 | P | 483A | operands |

Syntax:

    <sop> Cu,Cv,Cw,Cx,Cy,Cz

    where:
    <sop> is a sexadic operator

Operation:

    <sop> Cu,Cv,Cw,Cx,Cy,Cz --> Cz

Description:

    The operation corresponding to <sop> is applied to the operands and the result stored in Cz.

Example:

            drawln   r(0:0),r(0:1),r(0:2),r(0:3),r(0:4),d(0:0)

Addressing Modes:

    Cu: r,d
    Cv: r,d
    Cw: r,d
    Cx: r,d
    Cy: r,d
    Cz: r,d

Format:

    | 0250 | P | 483B | operands |

Syntax:

　　<sop> Ct,Cu,Cv,Cw,Cx,Cy,Cz

　　where:
　　<sop> is a sexadic operator

Operation:

　　<sop> Ct,Cu,Cv,Cw,Cx,Cy --> Cz

Description:

　　The operation corresponding to <sop> is applied to the operands and the result stored in Cz.

Example:

　　　　　cpfnt　r(0:0),r(0:1),r(0:2),r(0:3),r(0:4),d(0:0),d(0:1)

Addressing Modes:

　　Ct: r,d
　　Cu: r,d
　　Cv: r,d
　　Cw: r,d
　　Cx: r,d
　　Cy: r,d
　　Cz: r,d

Format:

　　| 0250 | P | 483C | operands |

Syntax:

    <oop> Cs,Ct,Cu,Cv,Cw,Cx,Cy,Cz

    where:
    <oop> is a octadic operator

Operation:

    <oop> Cs,Ct,Cu,Cv,Cw,Cx,Cy,Cz --> Cz

Description:

The operation corresponding to <oop> is applied to the operands and the result stored in Cz.

Example:

        invfnt r(0:0),r(0:1),r(0:2),r(0:3),r(0:4),r(0:5),r(0:6),d(0:1)

Addressing Modes:

    Cs: r,d
    Ct: r,d
    Cu: r,d
    Cv: r,d
    Cw: r,d
    Cx: r,d
    Cy: r,d
    Cz: r,d

Format:

    | 0250 | P | 483D | operands |

Syntax:

    <oop> Cr,Cs,Ct,Cu,Cv,Cw,Cx,Cy,Cz

    where:
    <oop> is a octadic operator

Operation:

    <oop> Cr,Cs,Ct,Cu,Cv,Cw,Cx,Cy --> Cz

Description:

    The operation corresponding to <oop> is applied to the operands and the result stored in Cz.

Example:

        invfnt r(0:0),r(0:1),r(0:2),r(0:3),r(0:4),r(0:5),r(0:6),d(0:0),d(0:1)

Addressing Modes:

    Cr: r,d
    Cs: r,d
    Ct: r,d
    Cu: r,d
    Cv: r,d
    Cw: r,d
    Cx: r,d
    Cy: r,d
    Cz: r,d

Format:

    | 0250 | P | 483E | operands |

Syntax:

move <ea1>,<ea2>

where:
<ea> must be one of the addressing modes listed below

Operation:

source --> destination

Description:

The value found at the source address is copied into the destination address.

Example:

move r(0:0),d(0:0)
move d(0:1),r(0:4)
move r(0:0),data
move {addr,data},r(0:20)
move {int,100},r(0:20)@
move r(0:20)@10,r(0:10)

data: ds 100

Addressing Modes:

<ea1>: d,r,ri,rid,ridn,m,pcr,i

<ea2>: d,r,ri,rid,ridn,m,pcr

Format:

| 0250 | P | { | H850 |...| H884 | } | operands |

Syntax:

   push <ea>,S

   where:
   <ea> is one of the addressing modes listed below

Operation:

   source --> S

Description:

   The source value is pushed onto stack S. The programmer has no access to the stack pointer.

Example:

                  push {int,100},s(0)
                  push r(0:10),s(1)
                  push d(0:0),s(1)

Addressing Modes:

   <ea>: d,m,pcr,r,ri,rid,ridn,i

   S: s

Format:

   [0250] [P] { [H880]...[H887] } [operands   ]

Syntax:

pop S,<ea>

where:
<ea> is one of the addressing modes listed below

Operation:

S --> destination

Description:

The source value is popped off stack S and stored at <ea>. The programmer has no access to the stack pointer.

It is an error to attempt to pop a value from an empty stack.

Example:

        pop s(0),r(0:1)
        pop s(0),data
        pop s(1),d(0:0)

data: ds 1

Addressing Modes:

S: s

<ea>: d,m,pcr,r,ri,rid,ridn

Format:

$\boxed{0250}$ $\boxed{P}$ { $\boxed{H889}$...$\boxed{H88f}$ } $\boxed{\text{operands}}$

Syntax:

   popx S

Operation:

   S -->

Description:

   The top value of stack S is removed.

   It is an error to attempt to remove the top of an empty stack.

Example:

          popx s(0)

Addressing Modes:

   S: s

Format:

   | 0250 | P | 2888 | operands |

Syntax:

nop

Operation:

-

Description:

Does nothing.

Addressing Modes:

-

Format:

| 0250 | P | 18A0 |

Syntax:

   stop

Operation:

  -

Description:

  Execution is terminated.

Addressing Modes:

  -

Format:

  | 0250 | | P | | 18A1 |

Syntax:

jmp <ea>

where:

<ea> is one of the addressing modes listed below

Operation:

<ea> --> PC

Description:

Execution resumes at <ea>.

· If **jmp** follows a **rorg** directive, a jump to memory absolute is converted to a branch.

Example:

                jmp here
                jmp r(0:0)
here: jmp (1:150)@

Addressing Modes:

<ea>: m,r,mi,pcr

Format:

| 0250 | | P | { | H8A2 | ... | H85A4 | } | operands |

Syntax:

    bra <ev>

    where:
    <ev> is one of the addressing modes listed below

Operation:

    PC + <ev> --> PC

Description:

    Execution resumes at the sum of the program counter and the effective value.

Example:

        bra 100

Addressing Modes:

    <ev>: a,r

Format:

    | 0250 | | P | | H8A5 |...| H8A6 | } | operands |

Syntax:

    if R <rop> <ev>,M
    if <bop> <ea>,M

    where:
    <rop> is a relational operator
    <bop> is a test operator
    <ea> and <ev> are one of the addressing modes listed below

Operation:

    if R <rop> <ev> then
            M --> PC

    if <bop> <ea> then
            M --> PC

Description:

If the comparison is true, execution resumes at M; otherwise, with the next instruction.

Example:

                move   {int,10},r(0:0)
    loop: if   r(0:0) < {int,1},done
                sub   {int,1},r(0:0)
                jmp   loop
    done: if   int data,loop

    data   ds   1

Addressing Modes:

    R: r
    <ev>: r,i
    <ea>: r,m
    M: m,pcr

Format:

    |0250| |P|
    { |58A7|,|58A8|,|58AB|,|58AC|,|48AF|,|48B0|,|48B3|,|48B4| }
    |operands |

Syntax:

    if R <rop> <ev>,Mx,My
    if <bop> <ea>,Mx,My

    where:
    <rop> is a relational operator
    <bop> is a test operator
    <ea> and <ev> are one of the addressing modes listed below

Operation:

    if R <rop> <ev> then
        Mx --> PC
    else
        My --> PC

    if <bop> <ea> then
        Mx --> PC
    else
        My --> PC

Description:

    If the comparison is true, execution resumes at Mx; otherwise, at My.

Example:

```
             if     r(0:0) > r(0:1),case1,case2
stuff:       move   r(0:0),data
case1:       jsr    first,s(0)
             if     int r(0:0),case1
             stop
case2:       jsr    second,s(0)
             stop
```

Addressing Modes:

    R: r

    <ev>: r,i

    <ea>: r,m

    Mx: m,pcr

    My: m,pcr

Format:

$\boxed{0250}$ $\boxed{\text{P}}$

{ $\boxed{68A9}$, $\boxed{68AA}$, $\boxed{68AD}$, $\boxed{68AE}$, $\boxed{58B1}$, $\boxed{58B2}$, $\boxed{58B5}$, $\boxed{58B6}$ }

$\boxed{\text{operands} \quad}$

Syntax:

    jsr <ea>,S

    where:
    <ea> is one of the addressing modes listed below

Operation:

    PC --> S
    <ea> --> PC

Description:

    .The program counter is pushed onto stack S, and execution resumes at <ea>.

    Following a **rorg** directive, memory absolute is converted automatically to program counter relative.

Example:

          jsr   incr,s(0)

Addressing Modes:

    <ea>: m,mi,r,pcr S: s

Format:

    | 0250 | | P | { | H8B7 |...| H8B9 | } | operands |

Syntax:

    bsr <ev>,S

    where:
    <ev> is one of the addressing modes listed below

Operation:

    PC --> S
    PC + <ev> --> PC

Description:

    · The program counter is pushed onto stack S, and execution resumes at the sum of the program counter and <ev>.

Example:

               bsr   r(1:0),s(0)

Addressing Modes:

    <ev>: r,a S: s

Format:

    | 0250 | | P | { | 38BA |,| 38BB | } | operands · |

Syntax:

    rts S

Operation:

    S --> PC

Description:

    Execution resumes at the address popped from stack S.

Example:

incr:	add	{int,1},r(0:0)
	rts	s(0)

Addressing Modes:

    S: s

Format:

    | 0250 | | P | | 28BC | | operand |

Syntax:

    open S

Operation:

    S -->

Description:

    To open a file, four file parameters must be pushed on the stack, in proper order, before the open instruction is invoked. These attributes are: a string atom for the filename, a file descriptor atom, an integer atom for the access mode, and an integer atom for the data type (raw or AM typed values). The open instruction pops these parameters off the stack and opens the file. All future file operations are referenced by the file descriptor.

Example:

```
datafile        equ    {file,3}
                push   {string,"filename"},s(0)
                push   {datafile},s(0)
                push   {int,0},s(0)
                push   {int,0},s(0)
                open   s(0)
```

Addressing Modes:

    S: s

Format:

    | 0250 | P | 28C0 | operand |

Syntax:

    close S

Operation:

    S -->

Description:

    The file descriptor atom must first be pushed on the stack. The close instruction pops the stack and closes the file.

Example:

```
·datafile      equ    {file,3}
               push   {datafile},s(0)
               close  s(0)
```

Addressing Modes:

    S: s

Format:

    | 0250 | | P | | 28C1 | | operand |

Syntax:

    read S

Operation:

    S -->

Description:

    The file descriptor atom must first be pushed on the stack. The memory address atom for the destination buffer cell is pushed next. The read instruction pops these parameters off the stack and puts the next file cell in the destination buffer.

Example:

```
datafile      equ    {file,3}
              push   {datafile},s(0)
              push   {addr,data},s(0)
              read   s(0)
data          ds     100
```

Addressing Modes:

    S: s

Format:

    | 0250 | P | 28C2 | operand |

Syntax:

    write S

Operation:

    S -->

Description:

The file descriptor atom must first be pushed on the stack. The memory address atom for the source buffer cell is pushed next. The write instruction pops these parameters off the stack and puts the contents of the source buffer cell into the next file cell.

Example:

    datafile   equ   {file,3}
                  push   {datafile},s(0)
                  push   {addr,data},s(0)
                  write s(0)
    data   dc   {string,"hello world"}

Addressing Modes:

    S: s

Format:

    [0250] [P] [28C3] [operand   ]

LIST OF REFERENCES

Davis, D. L., *Portability and Reusability through Resource Abstraction*, Unpublished notes, Jan 1985.

Fasel, J., *Programming Languages as Abstract Data Types - Definition and Implementation*, Ph. D. Thesis, Purdue University, Aug 1980.

Foley, J. D., Van Dam, A., *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Massachusetts, 1984, pp. 23-163,593-622.

Goguen, J. A., Thatcher, J. W., Wagner, E. G. and Wright, J. B., *An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types*, Current Trends in Programming Methodology IV, Data Structuring, R. T. Yeh, ed., Prentice-Hall, Englewood Cliffs, N. J., 1978, pp. 80-149.

Goldberg, A., Robson, D., *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Massachusetts, 1983, pp. 329-362.

Griffin, R., *An Algorithm to Test for Confluence in a System of Left to Right Rewrite Rules*, Master's Thesis, Naval Postgraduate School, Monterey, Ca., Dec 1984.

Guttag, J. V., Horowitz, E. and Musser, D. R., "Abstract Data Types and Software Validation", *Comm. ACM, Vol. 21, No. 12*, Dec 1978, pp. 1048-1064.

Guttag, J. V., Horowitz, E. and Musser, D. R., *The Design of Abstract Type Specifications*, Current Trends in Programming Methodology IV, Data Structuring, R. T. Yeh. ed., Prentice-Hall, Englewood Cliffs, N. J., 1978, pp. 60-79.

Guttag, J. V., "Notes on Type Abstraction", *IEEE Transactions on Software Engineering*, Jan 1980.

Lilly, N., *An Algebraic Specification Language and a Syntax Directed Editor*, Master's Thesis, Naval Postgraduate School, Monterey, Ca., Dec 1984.

Myers, G. J., *Advances in Computer Architecture* (2nd Edition), Wiley, New York, 1982, pp. 17-29.

Yurchak, J., *The Formal Specification of an Abstract Machine, Design and Implementation*, Master's Thesis, Naval Postgraduate School, Monterey, Ca., Dec 1984.

Naval Postgraduate School, Tech. Report NPS52 84-022, *A Formal Method for Specifying Computer Resources in an Implementation Independent Manner*, by Davis, D. L., Monterey, Ca., Nov 1984.

INITIAL DISTRIBUTION LIST